Rodivaldo Marcelo Raimundo

Curso básico de programação em

Korn-Shell Script

Para usuários e administradores
de sistemas Unix e Linux

O que é o Korn-Shell
Variáveis de memória
Uso das aspas no Unix
Estruturas de controle
Desvios simples e múltiplos
Tratamento de arquivos de dados
Introdução a linguagem AWK
Comandos do Unix úteis em scripts
Links da Internet sobre Unix

Revisão Jan/2022

"Nenhuma linguagem de programação é perfeita. Não existe mesmo nenhuma que seja a ideal. Existem apenas linguagens adequadas, ou até mesmo inadequadas, para fins específicos."

- Herbert Georg Mayer, engenheiro da Intel, autor e professor sobre linguagens de programação.

"... existem cantos escuros no Bourne Shell, e as pessoas usam todos eles."

- Chet Ramey, mantenedor do GNU Bourne Again Shell (BASH).

". . . a atividade intelectual associada ao desenvolvimento de software é basicamente uma atividade de obtenção de insights."

- Stowe Boyd, tecnólogo da informação especializado em ferramentas de colaboração em tempo real(RTCT). Criador do termo 'hashtag' em 2007.

Curso básico de programação em Korn Shell-script

INTRODUÇÃO **7**

CAPÍTULO 1 - O QUE É O SHELL **8**

Intrínsecos do Shell x comandos do Unix 9
Shells comumente usados 9
Características do Korn Shell 10
O uso de *alias* 10
Edição da linha de comando 11
Ambiente do usuário 12
Configurando variáveis do Shell 12
Regras para variáveis do Shell 12
Duas variáveis importantes – PATH e TERM 13
Exercícios 14

CAPÍTULO 2 - CAPACIDADES DE SUBSTITUIÇÃO DO SHELL **15**

Armazenamento das variáveis do Shell 16
Criação e deleção de variáveis do Shell 16
Substituição de variáveis 17
Substituição de comando 17
Substituição do ~ (til) 18
Exercícios 19

CAPÍTULO 3 - INTRODUÇÃO AO QUOTING **20**

Caracteres usados no mecanismo de Quoting 20
Quoting Barra Invertida - \ 20
Quoting Aspas Simples - ' ' 21
Quoting Aspas Duplas - " " 21
Exercícios 23

CAPÍTULO 4 - O QUE É UM SHELL-SCRIPT **24**

Exemplo de Shell-script 25
Comentários em um Shell-script - # 25
Passando dados para um Shell-script 25
Parâmetros posicionais 26
Variáveis especiais do Shell - # e * 27
O comando `shift` 28
O comando `read` 29
O comando `print` e os caracteres de saída 30
Escrevendo funções no Shell 30
Uso de variáveis locais e globais em funções 31
Técnicas adicionais de programação 35
Comentários *35*
Definindo o interpretador de comandos *35*
Depurando (debugging) Shell-scripts *35*
Quebrando linhas grandes em mais de uma linha *36*
Constantes x variáveis: questão de bom senso *37*
Exercícios 39

CAPÍTULO 5 - CÓDIGOS DE RETORNO E A VARIÁVEL ' ? ' **40**

Avaliando expressões - `[[ ]]` 41
Testes numéricos *41*
Testes de string *42*
Comparação numérica x comparação de strings 43
Teste de arquivos 43
Operadores booleanos para o comando `test` 43
Comandos `exit` e `return` 45
Desvio condicional - a construção `if/else` 46
Desvio condicional - a construção `if/elif/else` 47
Desvio condicional - a construção `case` 47
Exercícios 49

CAPÍTULO 6 - REPETINDO A EXECUÇÃO DE UM GRUPO DE COMANDOS (LOOPING) .. **50**

Avaliação aritmética - `let` ou `(( ))` 51
Executando enquanto a condição seja verdadeira - `while` 52
Mais exemplos de uso da construção `while` 53
Executando até que a construção seja verdadeira - `until` 53
Mais exemplos de uso da construção `until` 54
Executando grupos de comandos através de uma lista - `for` 55
Montando um menu com opções numeradas - `select` 56
Alterando a sequência de execução de um loop - `break` e `continue` 59
Exemplo de uso de `break` e `continue` 59
Exercícios 61

CAPÍTULO 7 - CONCEITO DE SINAIS **62**

O que é uma TRAP? 63
O comando `trap` 63
Ignorando sinais 63
Localização do comando `trap` 63
Exercícios 65

CAPÍTULO 8 - UMA PEQUENA INTRODUÇÃO À LINGUAGEM AWK **66**

Encontrando todos os padrões procurados em uma linha. 68
Mais exemplos de uso da linguagem AWK e suas declarações 68
Arquivos de dados usados nos exemplos 69
Arquivo `BBS-list` *69*
Arquivo `inventory-shipped` *69*
Iniciando com o `awk` 70
Como são executados os programas pelo `awk` 70
Um exemplo bastante simples 70
Um exemplo com duas regras de pesquisa 71
Programas `awk` executáveis 72

CAPÍTULO 9 – COMANDOS DO UNIX ÚTEIS PARA A PROGRAMAÇÃO SHELL-SCRIPT . **73**

Obtendo o comprimento de uma string - `${#VARIÁVEL}` 74
Extraindo subcadeias de caracteres de uma string 74
Removendo espaços em branco - `tr -s " "` 76
Trocando as letras - `tr 'a-z' 'A-Z'` e `tr 'A-Z' 'a-z'` 77

Colocando o cursor em uma determinada posição - tput cup ... 78
Definindo um formato de saída para os dados - comando printf ... 79
Alinhando strings à direita ou à esquerda ... 79
Preenchendo strings com zeros à esquerda ... 80
Efetuando uma pausa no processamento do script - sleep ... 80

CAPÍTULO 10 - EXEMPLOS DE SCRIPTS-SHELL ÚTEIS ... 82

Limitando a quantidade de sessões simultâneas dos usuários - unico_unico.sh ... 83
Calculando a diferença entre intervalos de tempo - diftempo.sh ... 85
Criação de variáveis armazenando comandos de cores ANSI – colors.sh ... 88
Pontuando valores maiores que 999 - pontuar.sh ... 89
Total de memória consumida por usuários – chk_mem_by_user.sh ... 92
Usuários em estado ocioso por mais de 5 minutos - check_idle.sh ... 94

APÊNDICE A - ENDEREÇOS INTERNET REFERENTES A UNIX – JAN/2022 ... 96

APÊNDICE B - COMANDOS UNIX ÚTEIS EM SCRIPTS ... 97

bc - Calculadora binária ... 97
cut - Exibição de campos ou colunas ... 99
expr length – Retorna o tamanho de uma string ... 101
expr substr – Exibe partes de uma string ... 103
printf – Exibe dados (strings ou numéricos) formatados ... 104
set -u/+u - Ativa ou desativa o uso de variáveis não existentes ... 106
sleep - Faz uma pausa em segundos ... 107
tput cup – Posicionamento do cursor em tela ... 108
tput ed - Limpa até o final da tela ... 109
tput el - Limpa até o final da linha ... 110
tput blink - Coloca o cursor em modo piscante ... 111
tput bold - Coloca o cursor em modo negrito ... 112
tput smso - Ativa o modo vídeo reverso ... 113
tput rmso - Desativa o modo vídeo reverso ... 114
tr -s " " - Remove o excesso de espaços ... 115
tr "[a-z]" "[A-Z]" - Converte caracteres em maiúsculas ... 116

APÊNDICE C - RESPOSTAS DOS EXERCÍCIOS PROPOSTOS ... 117

Exercícios do Capítulo 1 ... 118
Exercício 1 ... *118*
Exercício 2 ... *118*
Exercício 3 ... *118*
Exercícios do Capítulo 2 ... 119
Exercício 1 ... *119*
Exercício 2 ... *119*
Exercício 3 ... *119*
Exercício 4 ... *119*
Exercício 5 ... *119*
Exercício 6 ... *119*
Exercícios do Capítulo 3 ... 120
Exercício 1 ... *120*
Exercício 2 ... *120*

Exercício 3 ..... *120*
Exercício 4 ..... *120*
Exercício 5 ..... *120*
Exercício 6 ..... *120*
Exercícios do Capítulo 4 121
Exercício 1 ..... *121*
Exercício 2 ..... *121*
Exercício 3 ..... *122*
Exercício 4 ..... *122*
Exercícios do Capítulo 5 123
Exercício 1 ..... *123*
Exercício 2 ..... *123*
Exercício 3 ..... *123*
Exercício 4 ..... *123*
Exercícios do Capítulo 6 125
Exercício 1 ..... *125*
Exercício 2 ..... *125*
Exercício 3 ..... *126*
Exercício 4 ..... *127*
Exercício do Capítulo 7 128
Exercício 1 ..... *128*
Exercício 2 ..... *128*

Introdução

É interessante como um ambiente operacional como o Unix pode possuir, embutido em um de seus interpretadores de comando, uma linguagem de programação completa, capaz de criar se for o caso, sistemas completos além de ferramentas que agilizam a operação e administração de sistemas.

Criado no final da década de 60, o Unix é um sistema operacional que começou como um ambiente de teste por estudantes que eram programadores e precisavam de um sistema operacional em que ferramentas independentes pudessem ser combinadas para obterem os resultados desejados.

Com a implementação da linguagem de programação Shell, o sistema foi tornando-se mais e mais poderoso. Podia-se combinar os recursos do Unix em forma de programas, e não somente no modo interativo.

Com esta obra procuro suprir para aqueles que vivem no ambiente Unix uma ferramenta tanto de referência como um tutorial. Referência porque por mais que usemos a ferramenta e nos atemos a um grupo de comandos, sempre esquecemos alguma funcionalidade que nos será útil algum dia. Tutorial por que para aqueles que querem aprender a linguagem Korn Shell, seja como mais uma nova linguagem ou como a primeira, poderá não só desenvolver sua lógica em uma linguagem completamente estruturada, mas também poderá sentir a diferença entre desenvolver programas no próprio sistema operacional e apenas criar arquivos com sequencias de comandos, pois às vezes mesmo no prompt de comandos, sentimos a necessidade de comandos para manipulação de strings (sequência de caracteres), operações matemáticas, tratamento de variáveis, desvios, estruturas de laço (looping) e até mesmo colocar a saída de um comando em uma variável.

Procurei tornar esse livro o mais didático possível, de modo a ser entendido tanto pelos profissionais da área como por iniciantes no assunto. Coloquei ao longo do livro alguns exemplos, para que você possa ir se acostumando com o novo ambiente.

Complemento que, para iniciar neste livro você deve, como pré-requisito, conhecer o ambiente Unix em seu modo interativo, isto é, interagindo no prompt de comandos, embora no início do livro ocorra uma revisão rápida de conceitos do Unix e sobre o comportamento do Korn Shell em modo interativo. Mesmo assim, você deverá estar acostumado aos comandos básicos do Unix e, principalmente, ao uso do manual on-line pois, mesmo depois de anos e anos de uso do Unix, sempre tem uma opção ou parâmetro de algum comando que nunca conseguimos nos lembrar.

Espero poder, àqueles que estão à procura de uma obra sobre Korn Shell em português, ajudá-los a entender a linguagem e que, se for o caso, encontrar o caminho para vencer a resistência ao aprendizado e uso do Unix no que se refere à programação estruturada. Sugestões e críticas construtivas serão sempre bem vindas.

Bom aprendizado e boa sorte.

Rodivaldo Marcelo Raimundo
Assessor Certificado em Gerenciamento de T.I.
Auditor de Segurança da Informação
Especialista em Sistemas Operacionais Unix

Capítulo 1 - O que é o Shell

O Shell é um programa interativo que serve como interpretador de linha de comando com as seguintes funções:

- pesquisa por um comando e executa o programa a ele associado
- substitui os valores das variáveis do Shell pelo conteúdo a elas associados
- executa substituição de comandos
- completa nomes de arquivos a partir de caracteres de geração de nomes de arquivos (wildcards)
- manipula redirecionamento de entrada e saída e pipelines

Ele é responsável pela interface entre o usuário e o sistema operacional, traduzindo os comandos por ele digitados para serem executados pelo hardware.

O Korn Shell contém uma interface de programação interpretada incluindo testes lógicos, desvios e loops. Aos programas criados usando a interface programada do Unix damos o nome de Shell-script.

Intrínsecos do Shell x comandos do Unix

Intrínsecos, comandos internos ou builtins, são comandos construídos no próprio interpretador de comandos, isto é, não são comandos armazenados como arquivos em disco. Exemplos:

set, while, if, case, for, test, alias, echo

Os comandos externos, ou transientes, do Unix geralmente são localizados no diretório `/usr/bin`, ou `/bin` dependendo da versão do Unix, sendo encontrados através da variável `PATH`.

Para descobrir se um comando é interno ou transiente, utilize o comando **`type`** seguido do nome do comando:

```
$ type builtin
builtin is a shell builtin

$ type alias
alias is a shell builtin

$ type ls
ls is a tracked alias for /usr/bin/ls

$ type vi
vi is a tracked alias for /usr/bin/vi

$ type df
df is /usr/bin/df
```

Tanto os nomes de comandos quanto os intrínsecos não devem ser usados para nomes de scripts ou variáveis, pois são palavras reservadas do Shell.

Shells comumente usados

Embora possam existir vários tipos de Shell no Unix, este livro irá cobrir o Korn Shell, um padrão internacional de Shell para a maioria dos ambientes Unix que seguem as especificações determinadas pela interface POSIX, criada pelo IEEE (Institute of Electrical and Electronic Engeneers). Essa interface padroniza a programação nas várias plataformas Unix, o que permite a excelente portabilidade dos programas desenvolvidos em Shell-script.

Para sua informação, cito abaixo os Shells mais comuns, sua descrição e o prompt que eles possuem quando em linha de comando:

Shell	Tipo	Prompt
==========	==========	=====
/usr/bin/sh	Bourne Shell	$
/usr/bin/ksh	**Korn Shell**	**$**
/usr/bin/csh	C Shell	%
/usr/bin/bash	BASH Shell	$

Para ambiente Linux o Korn Shell pode ser instalado, assim como o BASH (Bourne Again SHell) pode ser instalado (caso não esteja) nas versões comerciais de Unix (IBM AIX®, HP-UX®, Oracle-Solaris®, entre outros. O Shell padrão do Linux é o BASH, que pode ser usado também para seu aprendizado de Korn Shell com poucas adaptações, ou às vezes nenhuma adaptações.

Se deseja praticar shell-script no sistemas MS-Windows®, recomendo o uso do CygWin ou ativar, a partir do Windows® 10, o Windows® Subsystem for Linux (WSL), recurso do sistema que permite executar um sistema de arquivos Linux, junto com ferramentas de linha de comando e aplicativos GUI do Linux, diretamente no Windows®.

Características do Korn Shell

O Korn Shell possui várias características, mas as destacadas aqui são as mais importantes para o entendimento da programação Shell:

- Uso de *alias* (apelido/atalho da memória) no comando
- Mecanismo de histórico de comandos
- Recuperação e edição da linha de comando
- Capacidades avançadas no uso do comando **`cd`**
- Capacidade de programação avançada

O uso de *alias*

Alias é um novo nome, ou apelido, para o comando. Por exemplo, se digitarmos o comando `ls` para listar os arquivos, é razoável pensar que estávamos executando o binário (programa `/usr/bin/ls`). É possível, mas também pode ser que um *alias* seja executado.

Para conseguir isso se usa o comando interno `alias`:

`alias` ***`nome=string`***

Um *alias* pode ser usado para abreviar longas linhas de comando ou fazer com que comandos se comportem diferente da execução padrão.

Se *`string`* contiver espaços, deverá ser digitada entre aspas (simples ou duplas).

Alias sempre são executados antes mesmo dos arquivos de comandos em disco. O Shell executa os comandos na seguinte ordem:
- Aliases
- Funções
- Built-in (intrínsecos)
- Palavras chave (Keywords)
- Arquivo de comando em disco

Quando atribuído interativamente, o *alias* ficará ativo até o logout da sessão de terminal. Para que um *alias* esteja disponível no momento do logon, deve-se colocá-lo no arquivo **`/etc/profile`** (caso queira que os *aliases* estejam disponíveis para todos os usuários), no arquivo **`.profile`** no diretório de login do usuário (ficando disponível somente para o usuário logado) ou no arquivo **`.kshrc`**, também no diretório de login do usuário. Exemplos:

```
alias cls=clear
alias ls='ls -logt'
alias lsf='ls -laF'
alias rm='rm -i'
alias del='rm -f'
alias dir=ls
```

Para que os *alias* sejam carregados a partir do arquivo **.kshrc** - arquivo lido a cada execução do comando **ksh** - deve ser adicionada a seguinte linha no arquivo **.profile** do usuário:

```
export ENV=$HOME/.kshrc
```

Deslogue e logue novamente para a alteração surtir efeito ou execute:

```
. $HOME/.profile
```

Edição da linha de comando

Permite a recuperação da última linha de comando para ser executada.

Usando os comandos do editor de textos **vi** a linha poderá ser editada antes da execução.

Para ativar o recurso execute:

```
set -o vi
```

Para que este recurso seja atribuído automaticamente no momento do logon, coloque o comando acima em seu arquivo **.profile** ou **.kshrc**.

Para recuperar a linha de comando, pressione a tecla **ESC** e em seguida a tecla **K**, que no editor **vi** tem a função seta-para-cima. Utilize os comandos padrão do vi para a edição da linha. Ao concluir, pressione ***[ENTER]*** para executar.

O editor de texto *vi* não é a mesma coisa que o *vim*. Não use as teclas de seta pois estas não funcionam no modo usual como em outros editores de texto. Use os comandos de navegação do vi. Se deseja navegar usando as teclas de seta, cheque a configuração do seu emulador de terminal texto (PuTTY, Moba etc).

Ambiente do usuário

O ambiente do usuário descreve a sessão para o sistema, contendo as seguintes informações, geralmente armazenadas em variáveis de ambiente:

- caminho para o diretório home - variável **HOME**;
- onde enviar seu correio eletrônico - variável **MAIL**;
- fuso horário no qual você está trabalhando - variável **TZ**;
- com que nome você se logou - variável **LOGNAME**;
- onde seu Shell pesquisará os comandos - variável **PATH**;
- seu tipo de terminal - variável **TERM**;
- outras definições que seus aplicativos possam precisar.

Muitos aplicativos necessitam que o ambiente seja customizado de alguma maneira. Isso é feito através da modificação do arquivo **.profile** do usuário. para que a customização valha de maneira geral, isso deve ser definido no arquivo **/etc/profile**.

O arquivo /etc/profile só pode ser manipulado pelo administrador do sistema operacional - usuário root - ou outro usuário com permissões equivalentes.

Configurando variáveis do Shell

Uma variável nada mais é do que um nome simbólico associado a um endereço na memória RAM, podendo valores serem associado ou alterados. Variáveis do Shell são atribuídas da seguinte maneira:

NOME_DA_VARIAVEL=conteúdo

Isto pode ser digitado como uma linha de comando ou em uma linha de um Shell-script.

Não deve haver espaço antes ou depois do sinal de igual. Isso assegura que a atribuição seja feita corretamente, não sendo interpretada como um comando e seus argumentos separados por espaços. Caso o conteúdo da variável contenha espaços, este deverá ser atribuído entre aspas (simples ou duplas) ou serão mostradas mensagens de erro.

Não é obrigatório, mas recomenda-se sempre usar letras maiúsculas ao criar suas variáveis. Além de ser o padrão do Korn Shell (e também do BASH), fica mais clara a leitura de seus programas Shell e denotará profissionalismo de sua parte.

Regras para variáveis do Shell

Os nomes das variáveis do Korn shell podem começar com um caractere alfabético (a-z e A-Z) ou underscore (_). Podem conter caracteres numéricos, mas nunca iniciar por um número. Caso o faça, o Shell retornará uma mensagem de erro como exemplificado a seguir:

```
$ VAR="ConteudoEmTexto"   # Criacao correta
$
$ 1VAR="ConteudoEmNumero"  # Criacao invalida (começa com número)
ksh: 1VAR=ConteudoEmNumero: not found
```

```
$
$ _1VAR="ConteudoEmNumero"  # Criacao válida pois inicia com underscore
$
```

Há, realmente, nomes de variáveis que contêm apenas dígitos (**0-9**) ou caracteres especiais (**!**, **@**, #, **%**, *****, **?**, **$**). Estas são variáveis reservadas do Shell para parâmetros especiais, definidos internamente (built-in) no interpretador de comandos.

Duas variáveis importantes – PATH e TERM

PATH – relação de diretórios, usada pelo Shell para a pesquisa de comandos. Os diretórios a serem pesquisados devem estar separados por dois pontos (**:**). Exemplo:

```
export PATH=/usr/bin:/etc:/bin:/home/user1:
```

Apesar da comodidade, evite colocar o diretório corrente (.) na lista de diretórios da variável PATH. Além de potenciais execuções não desejadas de programas no diretório corrente e com resultados desastrosos, também é considerada uma falha gravíssima de segurança.

TERM – descreve o tipo de terminal. Deve ser atribuído um valor de acordo com o terminal em uso. Para a maioria dos aplicativos usa-se o valor **vt100**, mas pode assumir outros valores dependendo do fabricante do terminal, ou da necessidade para um aplicativo de um tipo de terminal específico. O valor **vt100** garante pelo menos um terminal de 80 colunas de largura por 24 linhas de altura. Exemplo:

```
export TERM=vt100
export TERM=vt320
export TERM=ibm3151
export TERM=linux
```

Exercícios

1 - Crie um alias chamado **`limpatela`** que execute o comando **`clear`**.

2 - Coloque o alias criado anteriormente no seu arquivo **`.kshrc`**. Faça com que seja carregado a cada execução de shell ou em cada login efetuado

3 - Altere o comportamento dos seguintes comandos de acordo com as solicitações abaixo:

- **`rm`** - deve pedir confirmação sempre que um arquivo seja removido, independente do uso dos caracteres geradores de nome de arquivo (**`*`**, **`?`** e **`[]`**)
- **`ls`** - deve mostrar lista longa, diferenciando, no nome de arquivo, diretórios, arquivos comuns, executáveis, links etc. Além disso, deve mostrar o *i-node* do arquivo listado
- **`clear`** - deve aguardar 5 segundos antes de limpar a tela

Capítulo 2 - Capacidades de substituição do Shell

Uma característica interessante no Shell é a capacidade de podermos manipular textos, números e até saída de comandos através de variáveis. A isso chamamos substituição do Shell.

Existem 3 tipos de substituição no Shell:

- substituição de variáveis
- substituição de comandos
- substituição do til

Esses métodos de substituição são utilizados para acelerar a execução e a digitação da linha de comando, mas serão bastante úteis quando usados no desenvolvimento de aplicativos em Korn Shell.

Armazenamento das variáveis do Shell

Variáveis são áreas de memória onde podem ser armazenados dados. Esses dados podem ser números, textos, listas de arquivos e até mesmo resultados da saída de comandos que podem ser utilizados posteriormente.

Existem duas categorias de variáveis ambientais que podemos definir no Unix:

- **Variáveis locais** – disponíveis somente para o Shell corrente, não sendo acessadas pelos subprocessos.
- **Variáveis globais** – disponíveis tanto para o Shell corrente quanto para os subprocessos que venham a usar seu conteúdo.

Não Alterar o valor de uma variável global em subprocessos não altera o valor da variável no processo pai.

Para tornar uma variável com escopo global, ela deve ser exportada usando-se o comando **`export`**.

Para a visualização das variáveis locais use o comando **`set`**. Para verificar quais variáveis estão exportadas (globais) use o comando **`env`**.

Será mostrado o conceito de variáveis locais em Shell-scripts, onde uma variável pode ter o mesmo nome em módulos de funções diferentes, mas podendo ter conteúdos distintos em cada função.

Criação e deleção de variáveis do Shell

O armazenamento de variáveis ocorre da seguinte maneira:

- **Variáveis locais** – criadas através da sintaxe ***`NOME=conteúdo`***. Possuem escopo local, isto é não estão disponíveis para os subprocessos.
- **Variáveis globais** – estão disponíveis para o Shell corrente e para seus subprocessos. Se já existirem, o seus conteúdos serão atualizados.

Lembre-se que para tornar uma variável global usa-se o comando **`export`**. Este comando permite criar a variável e, ao mesmo tempo, exportá-la. Veja o exemplo:

```
export ORACLE_SID=dt4p
```

Para tornar uma variável imune a alteração ou deleção deve-se usar o comando **`readonly`**:

```
readonly NOME_DA_VARIAVEL
```

Para apagar uma variável, desde que não seja somente-leitura, use o comando **`unset`**:

```
unset NOME_DA_VARIAVEL
```

Variáveis somente-leitura não podem ser apagadas. Elas somente deixarão de existir no momento em que for efetuado o logout da sessão de terminal, ou se o programa que criou a variável no subprocesso for encerrado.

Substituição de variáveis

Cada variável terá um valor a ela associado. Quando o nome de uma variável for precedido por um sinal de **$** (dólar) o Shell substituirá o parâmetro pelo conteúdo da variável. Este procedimento é conhecido como **Substituição de Variável**. Uma das maneiras de exibirmos o conteúdo de uma variável é usando o comando **print -**:

```
$ print - $PATH
/usr/bin:/usr/contrib/bin:/usr/local/bin
```

Ou pode ser passada como argumento para um comando:

```
$ ARQUIVO=/home/morro.txt
$ more $ARQUIVO
```

Pode-se também alterar uma variável utilizando a mesma e/ou outra variável:

```
PATH=$PATH:$HOME:$ORACLE_HOME
```

Preste atenção no exemplo de concatenação abaixo:

```
$ TXT1=Casa
$ TXT2=Mae
$ TXT3=Joana
$ print - $TXT1da$TXT2$TXT3
MaeJoana
$ print ${TXT1}da$TXT2$TXT3
CasadaMaeJoana
```

Observe que, no último exemplo, foram usadas as chaves para circundar o nome da variável, senão o Shell poderia interpretar a variável como **TXT1da** o que seria um nome de variável válido, diferente de **TXT1**. Neste caso a variável **TXT1da** não existe, retornando uma string nula para a concatenação.

Substituição de comando

A substituição de comandos é usada para trocar a saída de um comando por seu resultado dentro da mesma linha de comando. Isto é útil quando for necessário armazenar a saída de um comando em uma variável ou passar essa mesma saída para outro comando.

A sintaxe utilizada é:

$(*comando*)

A substituição de comandos permite que você capture o resultado de um comando e utilize-o como um argumento para outro comando ou o armazene em uma variável.

Veja os exemplos a seguir:

- Armazenando o diretório corrente em uma variável:

```
$ pwd
/home/rodivaldo.raimundo
$ DIR_ATUAL=$(pwd)
$ print - $DIR_ATUAL
/home/rodivaldo.raimundo
```

- Passando uma lista de arquivos do comando `ls` para o comando `cp`:

```
$ cp $(cat filelist.txt) /home/user1/backup
```

- Efetuando um backup dos arquivos do dia de hoje, criando um arquivo de backup com *timestamp* em seu nome:

```
tar -cvf backup.$(date +"%Y%m%d.%H%M%S").tar \
$(ls -la|grep -v ^d|grep "$(date +"%b %e")"|awk '{print $9}')
```

A barra invertida no final da linha informa ao shell que o comando continua na linha seguinte, o que gera um prompt secundário (**>**) para que a linha seja completada. Isto é muito útil quando a linha de comando digitada começa e estourar os limites visíveis da tela.

Substituição do ~ (til)

A substituição do til é executada de acordo com as seguintes regras:

- Um til seguido pelo nome de um usuário existente em **/etc/passwd**, assume **$HOME** daquele usuário

```
cd ~user10              - vai para o diretório /home/user10
ls -logtr ~user5/bin    - lista o conteúdo do diretório /home/user5/bin
```

- Um til sozinho ou em frente a uma / é substituído pelo conteúdo da variável `HOME`

```
HOME=/home/user3
print - ~               - retorna /home/user3
ls -lF ~/file1          - será substituído por /home/user3/file1
```

- Um til seguido do sinal de **+** é substituído pelo valor da variável `PWD`

```
PWD=/home/user3/tree
ls -logt ~+/poodle      - Será substituído por /home/user3/tree/poodle
```

- Um til seguido de **-** será substituído pelo valor da variável `OLDPWD`

```
OLDPWD=/home/user3/mail
ls ~-                   - Será substituído por /home/user3/mail
```

Exercícios

1 - Utilizando a substituição de comandos, atribua a data de hoje no formato ***dd/mm/yyyy*** à variável **`TODAY`**. Consulte o manual on-line para verificar quais o formatos possíveis para a exibição de data e hora.

2 - Liste o conteúdo do diretório de outro usuário usando a substituição do til?

3 - Crie uma variável chamado **`MYNAME`** e armazene nela o seu primeiro nome. Exiba o conteúdo armazenado.

4 - Faça com que os subprocessos (shells filhos) possam ter acesso à variável **`MYNAME`**?

5 - Torne a variável **`TODAY`** somente-leitura. Em seguida tente exclui-la ou alterar seu conteúdo.

6 - Modifique seu prompt de comando, de modo que ele fique com a aparência a seguir:

```
Nome_do_Usuário@Nome_da_Maquina:Diretório_Atual=>
```

Use substituição de comandos e de variáveis.

Capítulo 3 - Introdução ao Quoting

Muitos caracteres no sistema Unix tem significado especial para o Shell. Por exemplo:

- O espaço em branco é o delimitador entre comandos, opções e argumentos
- O *carriage-return* (CR) dá o sinal para o Shell executar a linha introduzida
- O **$** é usado para mostrar o valor associado a uma variável

Ha situações em que você quer que o Shell não interprete o significado especial associado a esses caracteres. Você precisa apenas do caractere literal.

O sistema Unix oferece um mecanismo que remove/omite o significado especial de determinados caracteres. Esse mecanismo é conhecido como ***Quoting***.

Caracteres usados no mecanismo de Quoting

Os caracteres usados no mecanismo de Quoting são os seguintes:

- Barra invertida - `\` - ignora o caractere subsequente
- Aspas simples - `' '` - ignora todos os caracteres envolvidos
- Aspas duplas - `" "` - ignora os caracteres envolvidos com exceções

Quoting Barra Invertida - \

A barra invertida (\) remove o significado do caractere situado imediatamente depois dela.

A barra invertida sempre remove o significado do próximo caractere sem exceções. Exemplos:

- Exibindo um texto:

```
$ print - a \\ omite o efeito do próximo caractere
a \ omite o efeito do próximo caractere
```

- Criação de variáveis

```
$ COR=vermelho\ branco\ e\ azul
$ print - o valor de \$COR e $COR
o valor de $COR e vermelho branco e azul
```

- Ignorando a mudança de linha, permitindo dividir linhas longas em mais de uma linha

```
$ print - um dois \
> tres quatro
um dois tres quatro
```

O prompt secundário (>) sempre aparecerá quando pressionada a tecla ENTER após \.

Quoting Aspas Simples - ' '

As aspas simples (' ') – não confundir com acento grave (` `) – também desativam a interpretação especial dos caracteres especiais.

Todos os significados dos caracteres especiais situados entre aspas simples são omitidos. As aspas simples não podem ter seu significado omitido porque são necessárias para abrir e fechar a string assinalada.

Exemplos de uso de aspas simples:

- Criação de variáveis

```
$ COR='vermelho branco e azul'
$ print - 'o valor de $COR e $COR'
o valor de $COR e $COR
```

- Exemplo com a correta substituição de variável

```
$ COR='Ciano, Fucsia, Roxo, Branco e Azul'
$ print - 'o valor de $COR e' $COR
o valor de $COR e Ciano, Fucsia, Roxo, Branco e Azul
```

- Não se remove o efeito da aspa simples diretamente entre aspas simples

```
$ print - 'this doesn't work'
> faltou fechar a segunda aspa'
this doesnt work
faltou fechar a segunda aspa
```

- Mostrando literalmente caracteres especiais do Unix

```
$ print - *****
arq_temp.001 arq_temp.002 arq_temp.003 arq_temp.004 arq_temp.005
$ print - '*****'
*****
$ print - ########### Comentarios nao sao mostrados

$ print - '########### Comentarios nao sao mostrados'
########### Comentarios nao sao mostrados
```

Quoting Aspas Duplas - " "

Aspas duplas (" ") são menos abrangentes.

A maior parte dos caracteres especiais situados entre aspas duplas perdem o significado. As exceções são:

- o símbolo **$** (quando usado para substituição de variáveis e de comandos)
- **{***nome de variável***}**
- a barra invertida (****)
- a crase (**`**)
- e a aspa dupla (**"**), que é exigida para fechar a string assinalada.

Você pode usar a barra invertida ou aspas simples entre as aspas duplas para omitir o significado de **$** ou **"**. Exemplos:

- Aspas duplas com atribuição e substituição de variáveis

```
$ COR="vermelho, amarelo branco e azul"
$ print - "o valor de \$COR e $COR"
o valor de $COR e vermelho, amarelo branco e azul
```

- Aspas duplas com substituição de comandos

```
$ LOGNAME=$(whoami)
$ DIR_CORR="$LOGNAME - seu diretorio atual e $(pwd)"
$ print - $DIR_CORR
Administrador - seu diretorio atual e /home/Administrador
```

- Aspas duplas com uso dos caracteres especiais do Unix

```
$ print "todos estao aqui, \\, ' ', \" \",{ } ( )"
todos estao aqui, \, ' ', " ",{ } ( )
```

Exercícios

1 - Utilizando o comando `print -` e apenas o Quoting **** que produza a seguinte saída:

$1 million dollars...and that's a very excellent bargain!

2 - Atribua a string do exercício anterior à variável **LONG_STRING** utilizando substituição de comandos ou *quoting*.

3 - O que acontece quando é digitado **banner good day** e **banner "good day"**?

Quantos argumentos foram passados ao comando **banner** em cada um dos casos?

4 - Considerando que a variável ABC possui o conteúdo **Korn Shell**, descreva o que acontece na execução das duas linhas de comando abaixo:

`print - '$ABC' :` _____

`print - "$ABC":` _____

5 - Armazene na variável **ASPA_DUPLA** o caractere **"**. Verifique se ocorreu corretamente a atribuição.

6 – Exiba as seguintes mensagens de maneira a obter-se a saída a seguir:

`Este anula o significado do caractere após ele: \`

`Estes anulam o de qualquer um por eles envolvidos: ' '`

`E estes anulam o de todos, exceto \, $, $(), ${} e ": " "`

Capítulo 4 - O que é um shell-script

O Shell é um interpretador de comandos. Ele interpreta os comandos que você digita no prompt do Shell. Entretanto, você pode ter um grupo de comandos para executar várias vezes.

O Shell disponibiliza a capacidade de armazenar esses comandos em um arquivo e executá-lo como qualquer outro programa que pertence ao sistema Unix. Este arquivo de comandos é conhecido como **Programa Shell** ou **Shell-script**.

Quando executado, o programa rodará sequencialmente, como se os comandos fossem digitados interativamente no prompt do Shell.

Para que o Shell acesse o programa e execute-o, ele deve estar apto a ler o arquivo de programa e executar cada linha. Portanto, as permissões do Shell-script devem ser de **leitura e execução**.

Para que o Shell encontre seu programa, você pode informar o caminho completo de sua localização ou ele deve estar localizado em um dos diretórios designados em sua variável **`PATH`**. Os usuários podem criar um diretório **`bin`** abaixo do diretório **`$HOME`** para armazenar os programas que eles desenvolveram e incluir **`$HOME/bin`** na variável **`PATH`**.

Shell-scripts bem complexos podem ser desenvolvidos já que a linguagem suporta variáveis, argumentos de linha de comando, entradas interativas, testes, desvios e loops.

Exemplo de Shell-script

Para criar um Shell-script, considere o seguinte exemplo:

- Usando o editor de textos `vi`, crie um programa chamado **`meu_shell.sh`**

```
$ vi meushell.sh
```

- Adicione as seguintes linhas ao seu programa

```
#!/usr/bin/ksh
# Este é o programa meu_shell.sh
date
ls -F
```

- Após salvar o arquivo, adicione a permissão de execução a ele;

```
$ chmod +x meu_shell.sh
```

- Para executar o programa, digite o nome dele no prompt do Shell;

```
$ ./meu_shell.sh
Sun Aug 20 15:29:32     2006
arq_temp.001  arq_temp.003  arq_temp.005
arq_temp.002  arq_temp.004  meu_shell.sh*
```

Se quiser evitar o uso de `./` para executar seu Shell-script, adicione o diretório corrente (`.`) à variável `PATH`, MAS LEMBRE-SE DO RISCO DE USAR ESSE MÉTODO. São apenas dois caracteres a mais que serão digitados.

Comentários em um Shell-script -

É recomendado a inserção de comentários no seu Shell-script para identificar e esclarecer o conteúdo do programa. Os comentários são precedidos pelo símbolo **`#`**. O Shell não executa nada após o **`#`**, podendo ser inserido em qualquer lugar da linha de comando.

Passando dados para um Shell-script

Uma forma de passar dados para um Shell-script é através do ambiente. Exemplo:

- Na linha de comando, é criada uma variável global

```
$ export COLOR="Verde e Azul"
```

- Após isso, é criado o programa `color_001.sh` com o seguinte conteúdo

```
#!/usr/bin/ksh
# ---------------------
# Programa color_001.sh
# ---------------------
print - Voce esta executando o programa: color_001.sh
```

```
print - O valor da variável COLOR e: $COLOR
```

- Depois de adicionada a permissão de execução, o programa é executado

```
$ chmod +x color_001.sh
$ ./color_001.sh
Você esta executando o programa: color_001.sh
O valor da variável COLOR e: Verde e Azul
```

É importante lembrar: quando um Shell-script é executado, todo o ambiente é duplicado antes da execução (***fork***) e depois é iniciada a execução do script na nova sessão (***exec***). Isto significa que o script está rodando no Shell filho, como um subprocesso do Shell onde ele foi iniciado.

Se quiser que o script execute na mesma sessão de Shell há duas maneiras. A primeira faz com que o script rode e ao terminar, automaticamente a sessão do Shell corrente será encerrada como se você tivesse dado o comando **`exit`**. Para isso use o comando **`exec`** antes do nome do script:

```
$ exec color_001.sh
```

Isso fará com que o Shell `color_001.sh` seja executado no Shell corrente, fazendo com que, ao seu término, a sessão de terminal seja encerrada.

Se quiser que um script seja executado no Shell corrente sem o encerramento da sessão de terminal (salvo se for executado um comando **`exit`** dentro do script, preceda o nome do Shell-script com um ponto (**`.`**) seguido de um espaço:

```
$ . color_001.sh
```

Desse modo o script será executado e não encerrará a sessão de terminal corrente exceto, como mencionado anteriormente.

Parâmetros posicionais

A maioria dos comandos do sistema Unix aceita argumentos nas linhas de comando, que normalmente:

- informam ao comando corrente os arquivos sobre os quais o comando deve operar

```
$ cp arquivo.001 arquivo.002 # Comando com dois argumentos
```

- especificam opções que estendam as capacidades do comando

```
$ ls -logtr # Comando com diversas opções
```

- ou simplesmente fornecem strings de texto

```
$ banner tudo bem
```

O suporte a argumentos na linha de comando também estão disponíveis em programas Shell. Eles representam um mecanismo conveniente para passar informações ao seu utilitário.

Quando você desenvolve um programa que aceite argumentos de linha de comando, pode passar nomes de arquivos ou diretórios que você quer que seu utilitário manipule, como é feito nos comandos do sistema Unix. Você também pode definir opções de linha de comando que permitirão o acesso às capacidades extensivas de seu Shell-script.

Os argumentos na linha de comando são referenciados no seu Shell-script através de variáveis especiais que são definidas em relação a uma posição do argumento na linha de comando. Esses argumentos são chamados de **Parâmetros Posicionais** porque o assinalamento de cada variável especial depende da posição do argumento na linha de comando. O nome dessas variáveis corresponde à sua posição numérica na linha de comando. Portanto, os nomes dessas variáveis são os números **0**, **1**, **2** e assim por diante até o último parâmetro passado. Seus valores são acessados da mesma forma que qualquer outro valor, ou seja, você deve referenciá-los como **$0**, **$1**, **$2** etc.

Após **$9** as chaves devem ser utilizadas (**${10}**, **${24}**), caso contrário o Shell interpretará `$10` como **$1** com um **0** após.

$0 sempre será o programa shell em execução.

A única desvantagem no desenvolvimento de um programa que aceita argumentos de linha de comando é que os usuários, não experientes com a filosofia do ambiente Unix, devem conhecer sua sintaxe e o que cada um dos argumentos da linha de comando representam.

Se você estiver desenvolvendo um Shell-script para ser usado por um usuário final sem conhecimento ou noções de informática ou sistemas operacionais, procure sempre fazer os programas na forma de menu de opções ou comandos que possam ser digitados diretamente na linha de comando, sem a necessidade de que ele precise memorizar diversas opções, pois mesmo que você desenvolva uma documentação, o usuário nem sempre terá a tempo para lê-lo (às vezes nem vai querer ler mesmo). Para ele o que importa para ele é a produtividade e não a complexidade usada pelo programador no desenvolvimento e na operação do programa.

Variáveis especiais do Shell - # e *

As variáveis especiais do Shell **#** e ***** permitirão uma grande flexibilidade no tratamento de listas de argumentos variáveis. Através de **$#** você sempre saberá quantos elementos foram entrados e poderá acessar a lista inteira através da variável **$***, não importando o número de argumentos existentes.

Observe que o comando em execução ($0**) nunca é incluído na lista variável de argumentos.**

Cada argumento da linha de comando ainda mantém sua identidade individual, por isso você poderá referenciá-los coletivamente através de **$*** ou individualmente através de `$1`, `$2`, `$3`, e assim por diante. Observe o exemplo a seguir:

- Programa que recebe argumentos passados pela linha de comando

```
#!/usr/bin/ksh
# ----------------------
# Programa argumentos.sh
# ----------------------
print - "Quantidade de argumentos da linha de comando: $#"
print - "Lista de argumentos da linha de comando: $*"
print - "O primeiro argumento da linha de comando e: $1"
print - "O ultimo argumento da linha de comando e:" $(print - $*|\
     cut -d" " -f$#)
```

- Execução do programa `argumentos.sh`

```
$ ./argumentos.sh -ltr -f teste1 ult_arg
Quantidade de argumentos da linha de comando: 4
Lista de argumentos da linha de comando: -ltr -f teste1 ult_arg
O primeiro argumento da linha de comando e: -ltr
O ultimo argumento da linha de comando e: ult_arg
```

Segue tabela com o resumo dos parâmetros/argumentos posicionais do Shell:

Argumento	*Função*
$0	O nome do próprio Script-shell
$1 a **$9**	Parâmetros posicionais passados ao Script-shell
${10} em diante	Parâmetros posicionais após o **$9** devem usar chaves - **{** e **}** sem espaços
$#	Quantidade de argumentos passados ao Script-shell
$*	Lista de argumentos passados ao Script-shell (**$0** não incluso, string única)
$@	Lista de argumentos passados ao Script-shell (**$0** não incluso, strings separadas)

O comando `shift`

O comando **shift** deslocará os argumentos de linha de comando nos parâmetros posicionais.

Após um comando **shift *n***, todos os parâmetros em **$*** são movidos para a esquerda ***n*** posições e **$#** é decrementado em ***n***. O valor padrão para ***n*** é 1.

O comando `shift` nunca afeta o parâmetro posicional `$0`.

Uma vez concluído o comando **shift**, os argumentos que foram excluídos da linha de comando são perdidos. Caso você necessite referenciá-los posteriormente no seu programa, será necessário gravá-los antes de executar o comando **shift**. Observe:

- Script que demonstra o uso do comando `shift`

```
#!/usr/bin/ksh
# ----------------------
# Programa color_005.sh
# ----------------------
ARGS_ORIG=$*
print - "Existem $# argumentos na linha de comando"
print - "Eles sao $*"
shift 2
print - "*** Existem $# argumentos na linha de comando ***"
print - "Eles sao $*"
print - "Os argumentos originais sao => $ARGS_ORIG"
```

- Execução com alguns argumentos passados

```
$ ./color_005.sh azul Amarelo laranja Verde Carmim
Existem 5 argumentos na linha de comando
Eles sao azul Amarelo laranja Verde Carmim
*** Existem 3 argumentos na linha de comando ***
Eles sao laranja Verde Carmim
Os argumentos originais sao => azul Amarelo laranja Verde Carmim
```

O comando `read`

O comando **`read`** é usado para absorver informações digitadas no terminal durante a execução do programa. Normalmente você pode direcionar o usuário, através do comando **`print`**, informando a respeito da entrada desejada.

O comando **`read`** fará uma pausa no script, aguardando a digitação do dado a ser entrado na variável e ser pressionada a tecla **`[ENTER]`** em seguida.

O comando **`read`** especificará uma lista de nome de variáveis em que os valores serão assinalados com os conteúdos que o usuário informar no prompt, sendo que esses conteúdos devem ser delimitados por espaços em branco. Exemplo:

```
#!/usr/bin/ksh
# --------------------
# Programa pedenome.sh
# --------------------
clear
print -  "Programa $0"
print -  '--------------------'
print -n 'Entre com o codigo desejado: '
read CODIGO
grep $CODIGO arquivo.dat
```

Exemplos da saída do programa:

```
$ ./pedenome.sh
Programa ./pedenome.sh
--------------------
Entre com o codigo desejado: 1
0001 Bill Gates
$
$ ./pedenome.sh
Programa ./pedenome.sh
--------------------
Entre com o codigo desejado: 04
0004 Linus Tovalds
$
$ ./pedenome.sh
Programa ./pedenome.sh
--------------------
Entre com o codigo desejado: [0-9]
0001 Bill Gates
0002 Steve Jobs
```

```
0003 Warren Buffet
0004 Linus Tovalds
```

Caso existam mais variáveis especificadas para o comando **`read`** do que palavras digitadas pelo usuário, as variáveis que sobrarem serão assinaladas com o valor conhecido como **NULL** (nada).

Se o usuário digitar mais palavras do que a quantidade de variáveis existentes, todos os dados restantes serão assinalados para a última variável da lista.

Depois de assinaladas, você pode acessar as variáveis da mesma forma que qualquer outra variável do Shell, isto é, usando o processo de substituição de variáveis.

O comando `print` e os caracteres de saída

Cada caractere de saída deve ser precedido por uma barra invertida (`\`) e estar entre aspas duplas (`" "`).

Os caracteres de saída são interpretados pelo comando `print` e não pelo Shell.

Caractere	Ação
`\b`	Backspace
`\c`	Suprime a terminação newline
`\f`	Alimentação de formulário
`\n`	Newline (LF – line feed)
`\r`	Carriage return (CR – carriage return)
`\t`	Tabulação
`\a`	Caractere de alerta (^G ou beep)
`\\`	Barra invertida
`\0nnn`	Caractere de código ASCII nnn (em octal)

Escrevendo funções no Shell

O comando **`function`** é usado para escrever programas Shell modulares.

Programação modular é o conceito de se criar códigos utilizados frequentemente em certa área (módulo) do shell-script para que este seja chamado quando necessário em vez de reescrever o mesmo código dentro do programa. Também é chamado de sub-rotina.

Todas as funções a serem definidas devem, por boa prática, serem colocadas no início do shell-script, pois assim elas estarão prontas para serem invocadas pelo módulo principal que fica localizado no final do script.

As sintaxes possíveis para a criação de uma função são as seguintes:

```
function nome_da_função
{
   instrução 1
   instrução 2
   instrução 3
   ....
   instrução n
}
```

ou

```
nome_da_função ()
{
   instrução 1
   instrução 2
   instrução 3
   ....
   instrução n
}
```

Exemplo de criação de uma função e sua chamada no Shell-script:

```
install ()
{
   print - "Arquivo de instalação: $1"
   chmod +x $1
   mv $1 $HOME/bin
   print - "Instalação completada"
}
print - "Digite o nome do arquivo: \c"
read arquivo
install arquivo
```

Uso de variáveis locais e globais em funções

A principal utilidade da programação modular é organizar o Shell-script de modo a facilitar sua manutenção, concentrando em determinados módulos a criação de variáveis, *aliases* de comandos, caminhos padrão para diretórios, funções genéricas entre outras utilidades.

Em relação às variáveis ambientais elas podem ter escopo **global** ou **local,** sendo usado ou não o comando **`export`**. Em se tratando de programação Shell-script o programador, ao desenvolver suas funções, pode sentir a necessidade de criar, em funções diferentes no mesmo programa, **variáveis que terão o mesmo nome mas que seus conteúdos serão diferentes enquanto o programa é executado**.

Poderia ocorrer, por exemplo, de um Shell-script ter as funções **`calc1`**, **`calc2`** e **`calc3`**. Em tais funções existir uma variável chamada **`TOTAL`** e cada uma armazenar um valor diferente. Em resumo, endereços de memória física diferentes, mas com o mesmo nome e, consequentemente, com conteúdo diferentes.

A capacidade de separar variáveis locais de globais com o mesmo nome em um Shell-script, ainda que tenham o mesmo nome em funções diferentes, vai depender do uso do comando **typeset** e de qual sintaxe de criação das funções será usada.

Se usada a sintaxe **Nome_da_Função()**, todas as variáveis são tratados como globais. Independente do uso do comando **typeset**, alterar a variável chamada **TOTAL** na função **calc1** vai alterar o conteúdo dessa mesma variável em todos os módulos do programa, inclusive no módulo principal se ele fizer referência a tal variável.

Variáveis usadas em argumentos posicionais também são globais com esse formato de função. Por exemplo, $0 sempre retornará o nome do programa principal e nunca o nome da função que estiver sendo executada no momento. De $1 em diante, argumentos passados no módulo principal serão sempre os mesmos dentro da função caso os argumentos posicionais não sejam alterados com a chamada da função.

Quando se usa a sintaxe **function Nome_da_Função**, o comando **typeset** permitirá que variáveis com o mesmo nome, mas em funções diferentes, sejam tratadas como variáveis independentes, isto é, alterar o conteúdo de **TOTAL** em **calc1** não afetará a mesma variável nos demais módulos do Shell-script (nem mesmo as variáveis de parâmetros posicionais).

Veja a seguir os exemplos que clarificam tal fato:

- Variáveis sempre globais com a sintaxe ***Nome_Da_Função*** **()**:

```
#!/usr/bin/ksh
# ----------------------------------------
# Programa: global_vars.sh
# Variaveis sempre globais no shell-script
# ----------------------------------------
calc1 ()
{
   typeset TOTAL
   TOTAL=$((12+12))
   print - "Conteudo de TOTAL em $0 = $TOTAL"
}

calc2 ()
{
   typeset TOTAL
   TOTAL=$((12*12+13))
   print - "Conteudo de TOTAL em $0 = $TOTAL"
}

calc3 ()
{
   typeset TOTAL
   TOTAL=$((12+12))
   print - "Conteudo de TOTAL em $0 = $TOTAL"
   print - '-------------------------------------------------'
   calc1
   calc2
```

```
    TOTAL=$((2*2*2*2*2*2*2*2*2))
    print - "Conteudo de TOTAL em $0 = $TOTAL"
    calc1
    calc2
}
# ----------------
# Modulo principal
# ----------------
TOTAL='Texto no modulo principal'
print - ""
print - '-----------------------------------------------'
print - "Valor de TOTAL em $0 = $TOTAL"
print - '-----------------------------------------------'
calc1
print - '-----------------------------------------------'
calc2
print - '-----------------------------------------------'
calc3
print - '-----------------------------------------------'
print - ""
print - "Valor de TOTAL em $0 = $TOTAL"
print - '-----------------------------------------------'
print - ""
```

- Execução de **global_vars.sh**:

```
-----------------------------------------------
Valor de TOTAL em global_vars.sh = >>>*** Texto no modulo principal ***<<<
-----------------------------------------------
Conteudo de TOTAL em global_vars.sh = 24
-----------------------------------------------
Conteudo de TOTAL em global_vars.sh = 157
-----------------------------------------------
Conteudo de TOTAL em global_vars.sh = 24
-----------------------------------------------
Conteudo de TOTAL em global_vars.sh = 24
Conteudo de TOTAL em global_vars.sh = 157
Conteudo de TOTAL em global_vars.sh = 512
Conteudo de TOTAL em global_vars.sh = 24
Conteudo de TOTAL em global_vars.sh = 157
-----------------------------------------------

Valor de TOTAL em global_vars.sh = >>>*** Texto no modulo principal ***<<<
-----------------------------------------------
```

- Variáveis sempre locais com a sintaxe **function *Nome_Da_Função***:

```
#!/usr/bin/ksh
# --------------------------------
# Programa: local_vars.sh
# Variaveis locais no shell-script
# --------------------------------
function calc1
{
```

```
    typeset TOTAL
    TOTAL=$((12+12))
    print - "Conteudo de TOTAL em $0 = $TOTAL"
}
function calc2
{
    typeset TOTAL
    TOTAL=$((12*12+13))
    print - "Conteudo de TOTAL em $0 = $TOTAL"
}
function calc3
{
    typeset TOTAL
    TOTAL=$((12+12))
    print - "Conteudo de TOTAL em $0 = $TOTAL"
    print - '-------------------------------------------------'
    calc1
    calc2
    TOTAL=$((2*2*2*2*2*2*2*2*2))
    print - "Conteudo de TOTAL em $0 = $TOTAL"
    calc1
    calc2
}
TOTAL='>>>*** Texto no modulo principal ***<<<'
print - ""
print - '-------------------------------------------------'
print - "Valor de TOTAL em $0 = $TOTAL"
print - '-------------------------------------------------'
calc1
print - '-------------------------------------------------'
calc2
print - '-------------------------------------------------'
calc3
print - '-------------------------------------------------'
print - ""
print - "Valor de TOTAL em $0 = $TOTAL"
print - '-------------------------------------------------'
print - ""
```

- Execução de **local_vars.sh**:

```
-------------------------------------------------
Valor de TOTAL em global_vars.sh = >>>*** Texto no modulo principal ***<<<
-------------------------------------------------
Conteudo de TOTAL em calc1 = 24
-------------------------------------------------
Conteudo de TOTAL em calc2 = 157
-------------------------------------------------
Conteudo de TOTAL em calc3 = 24
-------------------------------------------------
Conteudo de TOTAL em calc1 = 24
Conteudo de TOTAL em calc2 = 157
Conteudo de TOTAL em calc3 = 512
```

```
Conteudo de TOTAL em calc1 = 24
Conteudo de TOTAL em calc2 = 157
------------------------------------------------

Valor de TOTAL em global_vars.sh = >>>*** Texto no modulo principal ***<<<
------------------------------------------------
```

Técnicas adicionais de programação

Comentários
Documente os programas Shell precedendo um comentário com um sinal de numeral (**#**). O caractere **#** é utilizado para definir comentários no Shell-script. Será ignorado tudo que vier após o **#** até o final da linha.

Os comentários informam, aos outros e a você que estão lendo o programa, o propósito dos comandos que estão contidos nele.

Portanto, comente seu código. Isso torna mais fácil para os outros entender (e apreciar), e mais fácil para você manter.

Definindo o interpretador de comandos
Uma forma maneira para a execução de um Shell-script é:

ksh *programa_Shell*

Neste caso, o Shell-script não precisa ter a permissão de execução, tendo que obrigatoriamente possuir permissão para leitura. O comando chama um sub-Shell e o designa como interpretador de comandos para ser utilizado durante a execução. Isto também é útil quando você está rodando sob um Shell e deseja executar um Shell-script escrito em outra linguagem de comando Shell.

Especifique o interpretador de comandos que será utilizado pelo Shell-script através do comando **#!/usr/bin/*Shell_name*** como a primeira linha do Shell-script. Se você está atualmente rodando sob um Korn Shell interativamente, mas possui um script do C-Shell que gostaria de executar, a primeira linha do programa C-Shell deverá ser:

#!/usr/bin/csh

Depurando (debugging) Shell-scripts
Se desejar testar se seu programa shell possa ter algum erro de sintaxe sem que ele seja executado de fato, use a opção **-n** do comando. Ele não executará seu programa, mas retornará as mensagens de erro em caso de sintaxe errada:

```
$ cat -n testshell.sh
     1  #!/bin/ksh
     2
     3  print 'teste de programa shell'
     4  print 'Hoje e: '$(date +"%d/%m/%Y e sao %T"
     5  print '
$
$ ksh -n testshell.sh
testshell.sh[6]: no closing quote
$ print - $?
1
```

Note que apesar da numeração de linhas reais do arquivo, o Shell retorna números de linhas adicionais, significando que, internamente no interpretador, ocorrem processos que resultam em linhas adicionais à quantidade de linhas que você vê listado (um detalhe sempre a ser observado).

Após corrigir as falhas, os erros não mais serão mostrados:

```
$ cat -n testshell.sh
     1  #!/bin/ksh
     2
     3  print 'teste de programa shell'
     4  print 'Hoje e: '$(date +"%d/%m/%Y e sao %T")
     5  print ''
$
$ ksh -n testshell.sh
$ print - $?
0
```

Apesar de não existir um depurador para o Shell-script, cada linha do Shell pode ser vista com o seguinte comando:

```
$ ksh -xv testshell.sh
#!/bin/ksh

print 'teste de programa shell'
+ print 'teste de programa shell'
teste de programa shell
print 'Hoje e: '$(date +"%d/%m/%Y e sao %T")
+ date '+%d/%m/%Y e sao %T'
+ print 'Hoje e: 19/12/2021' e sao 20:47:54
Hoje e: 19/12/2021 e sao 20:47:54
print ''
+ print
$
```

Isto exibirá cada linha do Shell-script antes de executá-lo, permitindo que você veja como o Shell está executando a geração de nome do arquivo, a substituição de variáveis e substituição de comandos..

O modo de debug do Shell-script é usado não somente para localizar erros de sintaxe, como também para sanar erros de lógica.

Independente da linguagem em que foi desenvolvido, qualquer programa pode rodar sem apresentar erros de sintaxe, mas pode estar gerando resultados indevidos ou indesejados como consequência de erros de lógica.

Quebrando linhas grandes em mais de uma linha

Caso você esteja editando um programa e queira quebrar uma linha em várias, use a barra invertida no final dela. Isso diz ao Shell que a linha seguinte deve ser interpretada como sendo a continuação da linha anterior. Veja o exemplo:

```
# ----------------------
# Programa mesma_linha.sh
# ----------------------
print "Embora este programa esteja mostrando varias \
linhas, na verdade esta mensagem se mostrada na \
tela como se fosse uma unica linha, porque o Shell \
interpreta como se fosse somente uma linha."
print "Basta executa-lo para ver o resultado..."
```

Veja o resultado da execução notando que, os espaços adicionais no início de cada linha quebrada foi mantido na saída na hora da execução.

```
$ ./mesma_linha.sh
Embora este programa esteja mostrando varias linhas, na verdade esta
mensagem se mostrada na tela como se fosse uma unica linha, porque o
Shell interpreta como se fosse somente uma linha.
Basta executa-lo para ver o resultado...
```

Isto será útil no momento em que, na edição de um script, a sua digitação começar a sair da área visível da tela.

Linhas muito longas podem atrapalhar a posterior leitura do programa. Neste caso, usando o recurso do Shell de continuação de linha na linha seguinte, facilita a leitura do programa em si, especialmente quando se tem linhas com muitos pipelines (**|**) em sequência.

Constantes x variáveis: questão de bom senso
Evite usar constantes ou literais (sejam numéricos ou strings). Em vez disso, use nomes de variáveis significativos. Isso tornará o programa mais fácil de entender e permite fazer alterações e atualizações sem interromper o aplicativo.

Em vez de especificar um nome de arquivo no programa e repeti-lo por todo seu código:

```
if [[ -f /var/log/messages ]];then
   grep -i -E 'error|fail|panic' /var/log/messages
   print "Nao foram encontradas falhas em /var/log/messages"
fi
```

Utilize, por exemplo:

```
export LOG_FILE=/var/log/messages
if [[ -f $LOG_FILE ]];then
   grep -i -E 'error|fail|panic' $LOG_FILE
   print - "Nao foram encontradas falhas em $LOG_FILE"
fi
```

Fazendo como demonstrado no último exemplo, se o arquivo de log mudar de nome ou de caminho, basta apenas uma alteração na linha em que a variável **LOGFILE** foi definida.

... lendo o código-fonte do UNIX para o Bourne shell (/bin/sh). Fiquei chocado com a quantidade de algoritmos simples que poderiam se tornar enigmáticos e, portanto, inúteis, por uma escolha inadequada de estilo de código. Eu me perguntei: "Alguém poderia se orgulhar desse código?"

-Landon Noll – Cientista de computação aposentado.

Exercícios

1 - Crie um shell-script chamado **`prog01.sh`** que execute os seguintes passos:

- exiba uma mensagem de boas-vindas como comando **`banner`**
- defina uma variável chamada **`MYNAME`** contendo seu primeiro nome
- mostre o conteúdo da variável criada com uma mensagem ao usuário
- mostre a data e hora no formato ***dd-Nome_do_Mes-YYYY***
- mostre todos os usuários logados no sistema.

2 - Interativamente, direcione a saída do comando **`date`** para a variável **`DATE_VAR`**. Em seguida crie um Shell-script chamado **`exibedata.sh`** que mostrará o conteúdo da criada anteriormente.

3 - Se a linha de comando de um shell-script é

```
meu_script.sh abc def -d -4 +900 xyz
```

O que será estará armazenado nas seguintes variáveis?

`$#` _____

`$3` _____

`$7` _____

`$*` _____

`$0` _____

4 - Crie um shell-script que peça seu nome completo e o exiba em ordem inversa.

Capítulo 5 - Códigos de retorno e a variável ' ? '

Todos os comandos do Unix gerarão um código de retorno após a execução do comando.

Tal código de retorno é usado para determinar se o comando foi executado normalmente (retorno `0`) ou encontrou algum erro (retorno **`diferente de 0`**). **O código de retorno diferente de `0`** reflete o erro que foi gerado.

Erros de sintaxe geralmente ajustam o código de retorno para **1**. O comando **`true`** sempre retornará **0** e o comando **`false`** sempre retornará **1**.

Em outras linguagens (C por exemplo), os valores `1` = *Verdadeiro* e `0` = *Falso* para códigos de retorno. Em Shell-script a lógica é invertida, ou seja, *`0=Verdadeiro`* e *`diferente de 0=Falso`*. É importante este entendimento para as tomadas de decisões em programas Shell, e também para que, caso você esteja acostumado com outras linguagens de programação, leve tal detalhe em consideração.

A maioria das decisões de programação serão controladas pela análise dos códigos de retorno. O Shell define uma variável especial (**`?`**) que manterá o valor do código de retorno do último comando executado. Sempre será possível mostrar o código de retorno anterior com o comando:

```
$ print - $?
```

Na execução de testes condicionais, o código de retorno indicará se a condição era verdadeira (retorno **0**) ou falsa (retorno **diferente de 0**).

Exemplos:

```
$ true
$ print - $?
0

$ ls
$ print - $?
0
$ print - $?
0

$ false
$ print - $?
1

$ cp
cp: missing file operand
$ print - $?
1
$ print - $?
0
```

Avaliando expressões - `[[ ]]`

O comando **`[[ ... ]]`** avalia a expressão e ajusta o código de retorno. Sintaxe:

`[[` ***`expressão`*** **`]]`**

Para verdadeiro será retornado valor 0. Para falso o valor será diferente de 0

`[[ ]]` pode avaliar a condição de:

- Inteiros - valores sem ponto decimal
- Strings - cadeias de caracteres alfanuméricos
- Arquivos - diretórios e os diversos tipos de arquivo do Unix

Devem possuir espaços em branco após o **`[[`** e antes do **`]]`**

Testes numéricos

Sintaxe:

`[[` *número* **relação** *número* **`]]`**

Operadores relacionais para números:

- **`-lt`** menor que
- **`-le`** menor que ou igual a
- **`-gt`** maior que
- **`-ge`** maior que ou igual a
- **`-eq`** igual a
- **`-ne`** não-igual a

```
$ NUMERO=3
$ [[ $NUMERO -lt 7 ]]
print - $?
0

$ [[ $NUMERO -gt 7 ]]
print - $?
1
```

Quando testar o valor de uma variável Shell, você deve se precaver da possibilidade de que a variável possa conter NULL (nada). Note o que pode ocorrer se a variável **`NUMERO`** não foi criada:

```
[[ $NUMERO == 3 ]]
ksh: test: argument expected
```

Se **NUMERO** não recebeu um valor prévio, seu conteúdo será NULL. Quando o Shell executar a substituição de variável, o comando que tentará executar será:

```
[[  == 3 ]]
```

que não é uma situação completa de teste e está garantida a causa de um erro de sintaxe.

Um modo simples de contornar isto é colocar a variável que está sendo testada entre aspas duplas:

```
[[ "$NUMERO" == 3 ]]
```

Quando o Shell executar a substituição de variável, o comando que tentará executar será:

```
[[ "" == 3 ]]
```

Como regra geral, você deve adquirir o hábito de envolver todas as expressões `$NOME_DA_VARIAVEL` com aspas duplas para evitar substituições indevidas de pelo Shell.

Testes de string

O comando **`[[ ... ]]`** pode ser usado para comparar strings.

Os operadores de string são:

- **`[[ stringA == stringB ]]`** - verdadeiro - strings são idênticas
- **`[[ stringC != stringD ]]`** - verdadeiro - strings são diferentes
- **`[[ stringA > stringB ]]`** - verdadeiro (baseado na tabela ASCII)
- **`[[ stringA < stringB ]]`** - falso (baseado na tabela ASCII)
- **`[[ -z stringX ]]`** - verdadeiro se o comprimento da string é 0
- **`[[ -n stringY ]]`** - verdadeiro se o comprimento é diferente de 0
- **`[[ stringZ ]]`** - verdadeiro se o comprimento é diferente de 0

A aspas duplas também protegerão a avaliação da string se a variável possuir brancos. Observe este exemplo:

```
$ X="Sim nos iremos"
$ [[ $X == "sim" ]]
```

O comando acima causa erro de sintaxe pelo fato de ser interpretado pelo Shell como:

```
[[ Sim nos iremos == "sim" ]]

ksh: test: too many arguments
```

Caso sejam usadas as aspas duplas para envolver a variável:

```
$ [[ "$X" == "sim" ]]
```

A sintaxe acima agora está correta, sendo interpretada pelo Shell como:

```
[[ "Sim nos iremos" == "sim" ]]
```

Comparação numérica x comparação de strings

Tecnicamente, caracteres, sejam eles números ou letras, são tratados pelo Shell como strings (cadeias de caracteres).

O Shell tratará todos os argumentos como números quando executar testes numéricos, e todos os argumentos como strings quando executar testes com strings. Vejamos os exemplos:

- Comparação numérica (zeros à esquerda são ignorados). O exemplo abaixo retorna verdadeiro por serem numericamente equivalentes

```
$ X=03
$ Y=3
$ [[ "$X" -eq "$Y" ]]
$ print - $?
0
```

- Comparação de strings. O exemplo abaixo retorna falso por serem duas strings diferentes

```
$ X=03
$ Y=3
$ [[ "$X" == "$Y" ]]
$ print - $?
1
```

Teste de arquivos

Sintaxe:

```
[[ -opção nome_do_arquivo ]]
```

Há várias validações de arquivos. Uma relação parcial inclui:

- **-f** - Verdadeiro se o arquivo existe e pode ser lido
- **-w** - Verdadeiro se o arquivo existe e pode ser gravado
- **-x** - Verdadeiro se o arquivo existe pode ser executado
- **-d** - Verdadeiro se o arquivo existe é um diretório

Operadores booleanos para o comando `test`

- **||** - or (**OU** lógico)
- **&&** - and (**E** lógico)

- `!` - **NOT** (negação)
- `( condições )` - agrupamento de condições

Exemplos comparação de strings:

```
$ [[ "$ANS" == "y" || "$ANS" == "Y" ]]
$ print - $?
1
$ ANS="n"
$ [[ "$ANS" == "y" || "$ANS" == "Y" ]]
$ print - $?
1
$ ANS="Y"
$ [[ "$ANS" == "y" || "$ANS" == "Y" ]]
$ print - $?
0
```

Exemplos comparação de números:

```
$ [[ "$NUM" -gt 10 && "$NUM" -lt 20 ]] ; print - $?
1
$ NUM=21
$ [[ "$NUM" -gt 10 && "$NUM" -lt 20 ]] ; print - $?
1
$ NUM=13
$ [[ "$NUM" -gt 10 && "$NUM" -lt 20 ]] ; print - $?
0
```

Exemplos comparação de números:

```
$ ls -l /home/root/arquivo.dat # Arquivo inexistente
ls: cannot access '/home/root/arquivo.dat': No such file or directory
$ [[ -s /home/root/arquivo.dat && -r /home/root/arquivo.dat ]]
$ print - $?
1
$ touch /home/root/arquivo.dat # Novo arquivo criado (vazio)
$
$ ls -l /home/root/arquivo.dat # Arquivo com zero bytes
-rw-r--r-- 1 user1 user1 0 Jan 19 14:02 /home/root/arquivo.dat
$
$ [[ -s /home/root/arquivo.dat && -r /home/root/arquivo.dat ]]
$ print - $?
1
$ print - "Texto adicionado no arquivo" >>/home/root/arquivo.dat
$ ls -l /home/root/arquivo.dat # Arquivo agora com 28 bytes
-rw-r--r-- 1 user1 user1 28 Jan 19 14:04 /home/root/arquivo.dat
$ [[ -s /home/root/arquivo.dat && -r /home/root/arquivo.dat ]]
$ print - $?
0
```

O operador **!** é usado em conjunto com os outros operadores:

```
$ [[ ! -d /home/root/arquivo.dat ]] # Verifica se o diretório existe
$ print - $?
1
```

O exemplo seguinte serve para identificar se há quatro de argumentos na linha de comandos, e se o primeiro argumento é um **/m**, e se o último argumento da linha de comando é um diretório ou um arquivo cujo tamanho seja maior do que zero:

```
$ set /m /var Arq1 Arq2   # Definidos $1, $2, $3 e $4
$ [[ ( $# -eq 4 ) && ( "$1" = "/m" ) && ( -d "$2" || -s "$2" ) ]]
print - $?
0
$ set /m /var Arq1 Arq2 Arq9   # Definidos $1, $2, $3, $4 e $5
$ [[ ( $# == 4 ) && ( "$1" = "/m" ) && ( -d "$2" || -s "$2" ) ]]
$ print - $?
1
```

Comandos `exit` e `return`

O comando **exit** terminará a execução de um Shell-script e ajustará o código de retorno. Ele é normalmente ajustado para **0** para indicar a finalização com sucesso, e para não-zero para indicar uma condição de erro. Se nenhum argumento é utilizado, o código de retorno é ajustado para o do último comando executado antes de **exit**.

Ocasionalmente você precisa retornar de uma função com um status de saída. O comando **return** para a execução da função e retorna para a procedure chamada com um código de retorno. Se o código de retorno não for especificado, será aquele do último comando dentro da função (análogo ao comando **exit**).

Fique atento: quando `exit` é chamado dentro de uma função, o Shell-script é encerrado.

Quando `return` é chamado fora de uma função, ele age exatamente como o comando `exit`.

- Exemplo do uso do comando **exit**:

```
# ------------------
# Programa exit.prog
# ------------------
print - "saindo do programa agora..."
exit 99

./exit.prog
saindo do programa agora...
$ print - $?
99
```

- Exemplo do uso do comando **return**:

```
# -----------------
# Programa ret.test
# -----------------
rtn ()
{
   print - "Retorno da funcao ..."
   return 99
}
print - "Chamando funcao rtn"
rtn
print - $?

./ret.test
Chamando funcao rtn
Retorno da funcao...
99
```

Desvio condicional - a construção `if/else`

Usada para desvios de decisão simples.

Sintaxe:

```
if [[ condição ]];then
   comando1
   comando2
   comando3
   ...
   comandoN
fi
```

Exemplo:

```
if [[ -s /home/root/.profile ]];then
   print - "Arquivo existe...\a"
fi
```

O comando **[[...]]** é usado em uma construção **if** para controlar o fluxo de execução, mas qualquer comando pode ser usado, uma vez que todos os comandos do Unix geram um código de retorno, como mostrado no exemplo:

```
if grep kingkong /etc/passwd;then
   print - "King Kong encontrado..."
fi
```

A construção **if** também executa controle de programas quando erros são encontrados, como abaixo:

```
if [[ $# -ne 3 ]];then
   print - "Sintaxe incorreta..."
   print - "Uso: $0 arg1 arg2 arg3"
   exit 99
fi
```

Desvio condicional - a construção `if/elif/else`

Usada para desvios de múltipla decisão.

Sintaxe:

```
if [[ condição ]];then
   comandos
   ...
elif [[ condição ]];then
   comandos
   ...
elif [[ condição ]];then
   comandos
   ...
else
   comandos
   ...
fi
```

Exemplos:

```
if [[ $X -lt 10 ]];then
   print - "$X e menor que 10..."
elif [[ $X -gt 10 ]];then
   print - "$X e maior que 10"
else
   print - "$X e igual a 0..."
fi
```

Desvio condicional - a construção `case`

A construção **case** normalmente é usada para suportar interfaces de menu ou interfaces que tomarão alguma decisão baseada em várias opções de entrada do usuário. Sintaxe:

```
case VARIÁVEL in
     caso1) comandos
            ...
     ;;
     caso2) comandos
            ...
     ;;
     caso3) comandos
            ...
     ;;
     casoN) comandos
            ...
     ;;
         *) comandos
            ...
     ;;
esac
```

É comum encerrar todos os padrões case com um **`*)`** para gerar uma mensagem ao usuário e informá-lo que ele não deu uma resposta aceitável. Exemplo:

```
#!/usr/bin/ksh

UpperCase ()# Converte para MAIUSCULAS
{
    print - $1|tr 'a-z' 'A-Z'
}

LowerCase () # Converte para minusculas
{
    print - $1|tr 'A-Z' 'a-z'
}

print - '-------------------'
print - "  MENU DE OPCOES  "
print - '-------------------'
print - "d - data e hora"
print - "w - usuario atual"
print - "l - lista o diretorio corrente"
print - ""
print -n "Por favor entre com sua opcao: "

read OPCAO

OPCAO=$(LowerCase $OPCAO)

case $OPCAO in
     [d]) print - "Hoje e $(date +"%d/%m/%Y e sao %T")"
     ;;
     [w]) print - "Usuario atual nesta sessao: $(whoami)"
     ;;
     [l]) print - "Conteudo do diretorio atual:"
          ls -l
     ;;
       *) print - "Opcao invalida\a"
     ;;
esac

print - ""
```

Indentation/endentação: embora não seja obrigatório é uma boa prática, quando estiver usando estruturas de desvio, assim como as de loop que veremos adiante, que você recue 3 ou 4 espaços para dentro dos comandos que iniciam e terminam as estruturas.

Note que em todos os exemplos foi usada a técnica de endentação, facilitando a leitura do programa.

Se todas as estruturas fossem alinhadas à esquerda, você poderia ter problemas com a manutenção de seus scripts, principalmente quando você tiver várias estruturas **`while`**, **`for`**, **`if`** e **`case`** encadeadas (umas dentro das outras).

Exercícios

1 - Crie uma variável chamada **`X`** e teste se o valor dela é a string **`XYZ.`**

2 - Crie uma variável chamada **`Y`** e atribua um número a ela. Teste se o seu conteúdo é menor ou igual a 0.

3 - Crie um programa que peça a entrada de um valor e o armazene na variável **`Y`**. Exiba **`sim-verdadeiro`** se o número entrado for positivo e **`não-falso`** se ele for negativo.

4 - Escreva um programa que cheque se a quantidade de argumentos passados na linha de comando é igual e 3. Se não for, deve ser mostrada uma mensagem de erro.

Capítulo 6 - Repetindo a execução de um grupo de comandos (looping)

As construções de loopings permitem que você repita uma lista de comandos, e tal como nas construções de desvios, a decisão de continuar ou encerrar um looping estará baseada no código de retorno de um comando chave.

O comando **`test`** é frequentemente usado para controlar a continuidade de um loop.

Lembre-se de sempre usar a técnica de endentação para que facilite a leitura de seus scripts, tanto para você quanto para outra pessoa que possa ser incumbida de dar manutenção.

Avaliação aritmética - `let` ou `(( ))`.

Os loops são normalmente controlados pelo incremento de uma variável numérica. O comando **`let`** habilita o shell-script para usar expressões aritméticas. Esse comando permite inteiros extensos (long integer).

O uso de **`(( ... ))`** em volta da expressão substitui o uso de **`let`**. Os operadores reconhecidos pelo Shell no comando **`let`** são listados abaixo, em ordem decrescente de precedência.

Operador	Descrição
`!`	Negação lógica (not)
`* / %`	Multiplicação, divisão e resto
`+ -`	Adição e subtração
`< > <= >=`	Comparação relacional
`== !=`	Igual e diferente

Os parênteses podem ser usados para mudar a ordem de avaliação de uma expressão, como em:

```
(( X = X / ( Y + 1 ) ))
```

Note que as aspas duplas são necessárias para omitir o significado principal dos parênteses e dos espaços em branco.

Se preferir sempre colocar espaços para separar operandos e operadores dentro da expressão, as aspas duplas devem ser usadas com o **`let`** ou deve ser usada a sintaxe `(( ))`:

```
(( X = X + ( Y / 2 ) ))
```

Ao usar os operadores lógicos e relacionais, a variável de código de retorno do Shell (`?`) refletirá o valor de verdadeiro (`0`) ou falso (`1`) do resultado.

Executando enquanto a condição seja verdadeira - `while`

A estrutura de looping **`while`** repete o loop enquanto a condição for verdadeira.

Sintaxe:

```
while <condição>;do
    comandos...
    comandos...
    comandos...
done
```

Exemplo:

```
# Programa test_while.sh
X=1
while (( X <= 10 ));do
    print - "Ola....o valor de X e $X"
    (( X++ ))

    sleep 1
done
```

Cuidado com os loops infinitos dentro de uma estrutura `while`. São aqueles em que o comando de controle sempre retorna verdadeiro e o programa nunca encerra sua execução. Exemplo:

```
while (( 1 == 1 ));do
    print - "Olá...estou executando sem parar"
done
```

A única maneira de interromper o programa acima é através de `^C`. Caso seja um script rodando em background, principalmente aqueles iniciados pela crontab do Unix, apenas será possível encerrar o processo pelo comando:

```
kill -9 Número_do_Processo
```

Mais exemplos de uso da construção `while`

Exemplo 1:

```
# Repete enquanto ANS é sim
ANS=sim
while [[ "$ANS" == "sim" ]];do
    print - "Digite um nome: \c"
    read NOME

    print - $NOME >>file.nomes

    print - "Continua? "
    print - "Digite sim ou nao"
    read ANS
done
```

Exemplo 2:

```
# Repete enquanto há argumentos na linha de comando
while (( $# != 0 ));do
    if [[ -d $1 ]];then
       print - "Conteúdo de $1:"
       ls -F $1
    fi
    shift
    print - "Há $# itens na linha de comando..."
done
```

Executando até que a construção seja verdadeira - `until`

A estrutura de looping **`until`** repete o loop até que a condição seja verdadeira, ou seja, **executa enquanto a condição é falsa**.

Sintaxe:

```
until <condição>;do
    comandos...
    comandos...
    comandos...
done
```

Exemplos:

```
# Programa test_until.sh
X=1
until (( X > 10 ));do
    print - "Ola....o valor de X e $X"
    (( X++ ))
done
```

Cuidado com os loops infinitos dentro de `until`. São aqueles em que o comando de controle sempre retorna falso, tornando o programa interminável, como exemplificado com relação à estrutura `while`. Exemplo:

```
X=1
until (( X == 0 ));do
    print - "Olá...estou executando sem parar"
done
```

Mais exemplos de uso da construção `until`

Exemplo 1:

```
# Repete até ANS ser não
ANS=sim
until [[ "$ANS" = "nao" ]];do
    print -n "Digite um nome: "
    read NOME

    print - $NOME >>file.nomes

    print - "Continua? "
    print - "Digite sim ou nao"
    read ANS
done
```

Exemplo 2:

```
# Repete até que não haja argumentos na linha de comando
until (( $#  ==  0 ));do
    if [[ -d $1 ]];then
       print - "Conteúdo de $1:"
       ls -F $1
    fi
    shift
    print - "Há $# itens na linha de comando..."
done
```

Executando grupos de comandos através de uma lista - `for`

O loop **`for`** é uma construção muito flexível. Ela á capaz de circular qualquer lista que possa ser gerada. As listas podem ser facilmente criadas por substituição de comandos. Com pipes e filtros uma lista pode ser gerada a partir de qualquer comando ou arquivo.

Sintaxe:

```
for <VARIÁVEL> in <lista>;do
    comandos...
    comandos...
    comandos...
done
```

A construção for funciona assim:

a) a ***VARIÁVEL*** é assinalada com a string do primeiro item da lista

b) os comandos são executados;

c) a ***VARIÁVEL*** é assinalada com o próximo item da lista

d) os comandos são executados

e) assim continua até que o último item da lista tenha sido processado pela estrutura **`for`**.

Exemplos:

```
# Programa test_for.sh

for X in 1 2 3 4 5;do
    print - "2 x $X = \c"
    (( X = X * 2 ))
    print - $X
done
```

Saída gerada por `test_for.sh`

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

Envia e-mail para todos os usuários do arquivo `/etc/passwd`

```
for NAME in $(grep home /etc/passwd|cut -d: -f1);do
    mail $NAME < aviso.reuniao
    print - "Enviado arquivo aviso para $NAME"
done
```

Lista o conteúdo de um diretório

```
for FILE in *;do
    if [ -d $FILE ];then
       ls -F $FILE
    fi
done
```

Montando um menu com opções numeradas - `select`

O loop **`select`**, semelhante em sua construção ao loop **`for`**, é usado para a criação de menus numerados, gerados a partir de uma lista.

Cada opção possui um número que é atribuído automaticamente a cada opção da lista. O usuário deve digitar o número da opção desejada e pressionar a tecla ENTER.

Sintaxe:

```
select <VARIÁVEL> in <Lista_de_Opções>;do
    comandos...
    comandos...
    ...
done
```

A opção escolhida fica armazenada na variável ambiental **`REPLY`**. Esta pode, através das estruturas de tomada de decisão (**`if`** e **`case`**), ser usada para executar uma determinada atividade exigida pelo Shell-script desenvolvido.

O menu pode ser encerrado pelo pressionamento da combinação das teclas *CONTROL+D*, mas aconselha-se que seja programada uma saída mais elegante através do comando `break`, o mesmo usado para sair incondicionalmente dos loops `while`, `until` ou `for`.

O prompt do menu, que usa a variável **`PS3`** e lhe é atribuído o conteúdo **`#?`**, pode ser personalizado para mostrar a mensagem desejada de acordo com o contexto do menu apresentado. A lista de opções que fará parte do menu pode ser gerada através de substituição de variáveis, substituição de comandos ou mesmo listas estáticas a partir de um arquivo.

Quando é executada uma opção do menu, este não volta a ser automaticamente mostrado quando retorna da execução, mostrando apenas o prompt do menu. Basta pressionar ENTER novamente que o menu será reapresentado ou atribua `""` (null) para a variável `REPLY` após a estrutura `if/elif/else/fi` ou `case/esac`.

Veja o exemplo do menu, exemplificado anteriormente para a instrução de desvio **case**, agora modificado para funcionar dentro de um loop **select**:

```
#!/usr/bin/ksh

# Definicao do prompt a ser mostrado pelo menu
# com um linha em branco acima da mensagem
# ----------------------------------------
PS3="
Digite a opcao desejada => "


# Lista de opcoes do menu passados para
# o comando select
#-------------------------------------
select OPCAO in "Data e hora" \
                "Usuarios logado" \
                "Lista diretorio corrente" \
                "Sair" ;do

    # A variavel REPLY é criada automaticamente pelo
    # loop select armazenando o numero escolhido

    print - ""

    case $REPLY in
         1) date +"Hoje e %d/%m/%Y e sao %T"
         ;;
         2) whoami
         ;;
         3) ls -ltr
         ;;
         4) print - "Encerrando o script em 5 segundos..."
            sleep 5
            break
         ;;
         *) print - "Opcao $REPLY invalida... Redigite\a"
            sleep 2
         ;;
    esac

    print - ""

    REPLY=""
done

print - ""
print - "Programa encerrado. Obrigado"
print - ""
```

Note, no exemplo acima, que para uma saída mais elegante da estrutura select, dispensando o uso do *CONTROL+C* ou *CONTROL+D*, foi usado o comando **break** para finalizar o loop **select** e mostrar uma mensagem de encerramento do programa.

A seguir, é apresentado um exemplo prático de uso do comando **select**, onde um administrador de banco de dados Oracle pode selecionar a instância de banco de dados a ser acessada usando um menu de opções chamado através da função **set_sid *instância*** ou **set_sid** e escolher no menu a instância desejada.

Crie um script chamado, por exemplo, **set_sid.sh** e coloque o código, mostrado logo a seguir, de acordo com o ambiente existente.

Para acessar a função bastará digitar, no prompt do Unix, o comando **set_sid** e em seguida escolher a instância desejada.

```
# ------------------------------------------------
# Adicione a função abaixo em seu arquivo
# .profile (usuário do Oracle) para selecionar
# a instancia a administrar
# Altere a variável SID_LIST de acordo com a sua
# necessidade
# ------------------------------------------------

function set_sid
{
   ARGCOUNT=$#

   OLDSID=$ORACLE_SID
   if [[ $ARGCOUNT -eq 1 ]] ; then
      ORACLE_SID=$1
   else
      PS3="Informe a instancia Oracle desejada => "

      SID_LIST="PRD1 PRD2 PRD3 Nenhuma Cancelar"

      select option in $SID_LIST;do
        case $REPLY in
             1) ORACLE_SID=prd1
                break
             ;;
             2) ORACLE_SID=prd2
                break
             ;;
             3) ORACLE_SID=prd3
                break
             ;;
             4) ORACLE_SID="No SID"
                break
             ;;
             5) return 0
             ;;
       esac
       REPLY=""
```

```
      done
   fi

   if [[ ! "${ORACLE_SID}" = 'No SID' ]] ; then
      NEWHOME=$(dbhome $ORACLE_SID)
   else
      NEWHOME=
   fi

   # print - 'Oracle SID Anterior  - '$OLDSID
   # print - 'Oracle SID Nova      - '$ORACLE_SID
   # print - 'Oracle Home Anterior - '$ORACLE_HOME
   # print - 'Oracle Home Novo     - '$NEWHOME
   if [[ ! ${NEWHOME}'Z' = 'Z' ]] ; then
      if [[ ! ${ORACLE_HOME}'Z' = 'Z' ]] ; then
         reppath "$ORACLE_HOME/bin" "$NEWHOME/bin"
      else
         PATH=$PATH:${NEWHOME}/bin
      fi

      ORACLE_HOME="${NEWHOME}"
   else
      if [[ ! ${ORACLE_HOME}'Z' = 'Z' ]] ; then
         reppath "${ORACLE_HOME}/bin"
      fi
      ORACLE_HOME=

   fi

   print - Oracle SID agora e  ${ORACLE_SID}
   print - Oracle Home agora e ${ORACLE_HOME}

   PS1=$(whoami)@$(hostname)' ('${ORACLE_SID}') -->'
   PS3='#?'

   unset NEWHOME SIDLIST ARGCOUNT
}
```

Alterando a sequência de execução de um loop - `break` e `continue`

`break` *`[n]`* - termina a interação do loop e salta para o próximo comando após o enésimo **`done`**.

`continue` *`[n]`* - cessa a interação atual do loop e salta para o início da próxima interação enésima incluindo o loop.

Exemplo de uso de `break` e `continue`

```
while true;do
    print - "Digite o nome do arquivo a remover: \c"
    read FILE
```

```
    if [[ ! -f $FILE ]];then
       print - "$FILE não é um arquivo comum...\a"
       continue #Volta ao início do loop
    fi
    print - "Removendo $FILE..."
    rm $FILE
    break #Sai para a primeira linha após done
done
```

Exercícios

1 - Crie um programa que peça para o usuário digitar um nome qualquer na variável **`NOME`**. Ele só encerrará o programa quando for digitada a string **`sair`**.

2 - Crie um programa que peça 6 números e exiba a soma total deles.

3 - Crie um menu que possua 3 opções:

`d` - mostrar a data no formato ***dd/mm/yy***

`h` - mostrar a hora no formato ***hh:mm:ss***

`l` - usuários logados

Se for pressionada a tecla **ENTER** o programa deve ser encerrado. Crie duas versões de menu: uma com a estrutura **`while`** e outra usando a estrutura **`select`**.

4 - Crie um programa que rode em retaguarda. Ele deverá checar se o usuário root está logado.

Se estiver, deverá mostrar uma mensagem no terminal a cada 2 segundos com um bip.

Capítulo 7 - Conceito de sinais

Um sinal é um flag transmitido para um processo quando certos eventos ocorrem. Alguns dos sinais mais conhecidos do Unix são:

NULL	Saída do Shell (**^D** ou exit)
HUP	Hung-up - enviado para processos em retaguarda no logout
INT	Interrupção (**^C**)
QUIT	Quit (**^**) - gera arquivo core
KILL	Morte certa (não pode ser capturado ou ignorado)
TERM	Terminação por software

O comando **kill** transmite um sinal explícito para um ou mais processos.

Digite **kill -l** para obter uma lista dos sinais disponíveis no sistema operacional.

O que é uma TRAP?

Uma *trap* é uma maneira de capturar sinais transmitidos a um processo e executar alguma ação baseada no tipo de sinal que foi recebido.

A *trap* simplesmente espera por sinais enviados ao Shell, identifica-o, e então executa a ação associada. Uma *trap* pode ser definido para remoção de arquivos temporários no término de um programa ou pode definir sinais que devem ser ignorados durante um segmento sensível na execução do programa.

O comando `trap`

O comando **`trap`** pode ser usado em programas Shell para capturar um sinal antes que ele possa abortar um processo. Pode ser executada alguma ação adicional ou alternativa após a captura do sinal.

O comando **`trap`** é normalmente colocado no início de um Shell-script, mas ele pode estar em qualquer parte do script. As *traps* são assinaladas pelo Shell quando este lê o comando **`trap`**. As *traps* são acionadas quando um sinal relacionado é recebido.

Sintaxe:

```
trap 'comandos' sinal1 sinal2 ... sinaln
```

Exemplo:

```
# Sai do shell apenas por ^C, ^\ ou por kill -15
trap ' print - bye ; exit' 2 3 15
while (( 1 == 1 ));do
    print - "Hello..."
done
```

Ignorando sinais

O comando `trap` pode ser usado para ignorar determinados sinais quando este for recebido pelo processo.

Sintaxe:

```
trap '' sinal1 sinal2 ... sinaln
```

Veja este exemplo de script ignorando o sinal INT (equivalente ao pressionamento da tecla `^C`):

```
trap '' 2
while (( 1 == 1 ));do
    print - "Hello..."
    sleep 2
done
```

Localização do comando `trap`

Apesar de o comando `trap` ser usualmente posto no início de scripts, dependendo de sua necessidade ou segurança em algum processamento ele pode ser colocado de acordo com as sugestões a seguir:

- Coloque o comando **`trap`** no início do programa para gerenciar a remoção de arquivos temporários se o programa for abortado:

  ```
  trap 'rm /tmp/tempfile;exit' 2 3 15
  ```

- Coloque **trap** antes de seções sensíveis ou críticas de código para ignorar sinais, como por exemplo em um processamento de backup que não pode ser interrompido:

  ```
  trap '' 2 3 15
  ```

- Restaure **trap** para ação padrão quando uma seção de código sensível é completada:

  ```
  trap 2 3 15
  ```

Exercícios

1 - Escreva um Shell-script que emita uma mensagem na sua tela a casa 3 segundos com um bip. Torne este programa imune aos sinais **INT**, **HUP** e **TERM**.

Como você pode parar este programa se:

- Estiver rodando em primeiro plano: _____

- Estiver rodando em segundo plano: _____

2 - Faça com que as linha de comando abaixo fique imune aos mesmos sinais do exercício anterior:

```
nohup find ./|cpio -ocvdum >/dev/rmt0 &
nohup gunzip -dv backup_banco.tar.gz|tar -xvf - &
```

Dica: consulte a página de ajuda do comando nohup para ajudá-lo a solucionar qualquer problema na execução das linhas de comando acima.

Capítulo 8 - Uma pequena introdução à linguagem AWK

A linguagem AWK foi inventada em 1977. É um utilitário bastante útil para a manipulação e formatação de saída de strings, pois permite que dados sejam mostrados em um formato melhor para visualização. O nome (bastante estranho por sinal) é derivado das iniciais de seus três criadores, Aho, Kernighan, e Weinberger. Ele é um *stream editor* (editor de fluxo).

Pode-se passar dados para o AWK via pipeline, mas também ele pode manipular dados de um arquivo. Basicamente isto significa que ele pode fazer qualquer coisa e muito mais. Ele possui a capacidade de guardar contextos, efetuar comparações em muitas outras coisas que qualquer linguagem de programação faz. Por exemplo, ele não é limitado somente a trabalhar com linhas simples. Ele pode unir múltiplas linhas, isso se você fizer as coisas certas.

A forma mais simples de um programa AWK é um programa de uma linha somente como neste exemplo genérico:

```
awk '{faça alguma coisa}' arquivo_texto
```

Este "faça alguma coisa" pode ser uma simples declaração `print` ou algo muito mais complicado, atuando em cada linha recebida do arquivo texto passado como parâmetro. Programas feitos com o AWK tem uma semelhança com a sintaxe da linguagem C. Um exemplo simples, usando um pipeline com o comando `ls`:

```
$ ls -l $HOME|awk '{print $1,$3,$4,$9}'
total
-rw-r--r-- user1 group1 Datafile.dat
lrwxrwxrwx user1 group1 Documents
drwx------ user1 group1 Mail
drwxr-xr-x user1 group1 aix
-rw-r--r-- user1 group1 arquivo.dat
-rw-r--r-- user1 group1 global_vars.sh
-rw-r--r-- user1 group1 lsvg_p.output
-rwxrwx-wx user1 group1 procdia.sh
drwxr-xr-x user1 group1 scripts
-rw-r--r-- user1 group1 xxx.xxx
```

O exemplo acima irá exibira as primeira, terceira, quarta e nona colunas do comando `ls -l`, onde as colunas são definidas como "coisas delimitadas por espaço". Esses delimitadores podem ser caracteres de tabulação ou os espaço em branco.

No caso do comando `ls`, serão exibidas as colunas contendo tipo e permissões, proprietário, grupo e nomes dos arquivos. Note que foi usado o sistema de pipeline, isto é, passando a saída do comando `ls` para o comando `awk`.

Veja este exemplo específico:

```
$ awk '/#/ {print "Linha comentada encontrada"}' /etc/hosts
Linha comentada encontrada
Linha comentada encontrada
Linha comentada encontrada
```

```
Linha comentada encontrada
Linha comentada encontrada
```

O exemplo anterior irá procurar, dentro de cada linha do arquivo `/etc/hosts` o caractere numeral (`#`) em qualquer posição. Se pelo menos um caractere for encontrado, será exibida a mensagem **`Linha comentada encontrada`** para cada linha onde o caractere existir.

Encontrando todos os padrões procurados em uma linha.

O AWK poderá processar todos os padrões encontrados na linha corrente. O programa a seguir, processado no meu computador pessoal:

```
$ awk '
      /#/ {print FNR,"Comentario encontrado"}
      $1 == "#" {print FNR, "Comentário na primeira coluna"}
      /^# /  {print FNR,"Comentario no início da linha"}
    ' /etc/hosts1 Comentario encontrado
```

processou as expressões regulares e mostrou a seguinte saída:

```
1 Comentario na primeira coluna
1 Comentario no início da linha
2 Comentario encontrado
3 Comentario encontrado
3 Comentario na primeira coluna
3 Comentario no início da linha
4 Comentario encontrado
5 Comentario encontrado
5 Comentário na primeira coluna
5 Comentario no início da linha
6 Comentario encontrado
6 Comentário na primeira coluna
6 Comentario no início da linha
7 Comentario encontrado
7 Comentário na primeira coluna
7 Comentario no início da linha
8 Comentario encontrado
8 Comentário na primeira coluna
8 Comentario no início da linha
9 Comentario encontrado
9 Comentário na primeira coluna
9 Comentario no início da linha
$
```

Como exibido na saída anterior, cada linha é processada e, se o padrão for encontrado o número da linha onde foi encontrado o padrão (variável interna **FNR**) será exibida, juntamente com a mensagem respectiva ao padrão.

Mais exemplos de uso da linguagem AWK e suas declarações

Os exemplos a seguir são uma tradução do manual do AWK encontrado no endereço da Internet **www.gnu.org/savannah-checkouts/gnu/gawk/manual/gawk.html**, em Janeiro de 2022, onde você poderá achar mais detalhes e exemplos sobre o uso do AWK

Trata-se de um manual completo da linguagem e de seu uso, possuindo exemplos práticos. Através de sites de busca pode-se encontrar também o mesmo manual em formato Adobe Reader (PDF).

Arquivos de dados usados nos exemplos

Para que os resultados mostrados possam ser praticados por você, crie os arquivos `BBS-list` e `inventory-shipped` como mostrados abaixo:

Arquivo *`BBS-list`*

```
aardvark      555-5553     1200/300          B
alpo-net      555-3412     2400/1200/300     A
barfly        555-7685     1200/300          A
bites         555-1675     2400/1200/300     A
camelot       555-0542     300               C
core          555-2912     1200/300          C
fooey         555-1234     2400/1200/300     B
foot          555-6699     1200/300          B
macfoo        555-6480     1200/300          A
sdace         555-3430     2400/1200/300     A
sabafoo       555-2127     1200/300          C
```

No arquivo `BBS-list`, cada registro contém o nome do computador do BBS, o número a ser discado, as taxas de transmissão (baud-rate) e um código para o número de horas em que ele estará disponível. Um **`A`** na última coluna significa que ele está disponível 24 horas por dia. Um **`B`** significa que o computador estará disponível somente à noite e nos finais de semana. Um **`C`** significa que o computador do BBS funcionará somente nos finais de semana.

Arquivo *`inventory-shipped`*

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
Dec  17  35  61 401

Jan  21  36  64 620
Feb  26  58  80 652
Mar  24  75  70 495
Apr  21  70  74 514
```

No arquivo `inventory-shipped` é representado o número de mercadorias enviadas durante o ano. Cada registro contém:

- o mês do ano
- a quantidade de engradados enviados

- a quantidade de caixas vermelhas enviadas
- a quantidade de sacos laranja enviados
- o número de pacotes azuis enviados naquele mês

Existem 16 entradas: 12 meses do ano e 4 meses do ano seguinte.

Iniciando com o `awk`

A função básica do `awk` é pesquisar em arquivos por linhas (ou outras unidades de texto) que contenham certos padrões. Quando uma das linhas coincide com o padrão, o `awk` executa uma ação específica naquela linha. Ele irá processar cada linha do arquivo até o processamento total do arquivo.

Um programa AWK consiste em uma série de regras. Inclusive pode-se definir módulos chamados funções, que é um tópico mais avançado que não será coberto nesse livro, porém há bastante detalhes no manual do AWK indicado anteriormente.

Sintaticamente, uma regra consiste em uma padrão a ser procurado, sendo que este será seguido por uma ação a ser executada quando o padrão for encontrado. As ações são incluídas entre chaves para que fiquem separadas do padrão sendo pesquisado. As regras são geralmente separadas por mudanças de linha. Portanto, um programa AWK se parece com a sintaxe abaixo:

```
padrão { ação }
padrão { ação }
...
```

Como são executados os programas pelo `awk`

Existem várias maneiras de se executar um programa `awk`. Se o programa é pequeno, é mais fácil incluí-lo na linha de comando, como no exemplo abaixo:

```
awk 'program' input-file1 input-file2 ...
```

Quando o programa é muito grande, é mais conveniente colocá-lo em um arquivo e executa-lo com um comando com a sintaxe abaixo:

```
awk -f program-file.awk input-file1 input-file2 ...
```

Um exemplo bastante simples

O programa a seguir faz uma pesquisa pela string de caracteres "`foo`" no arquivo `BBS-list`.

```
$ awk '/foo/ { print $0 }' BBS-list
```

Quando as linhas contendo a string `foo` são encontradas, elas são impressas, pois `print $0` para o AWK (não confundir com os parâmetros posicionais do Korn Shell) significa imprima a linha corrente.

Note que a expressão regular a ser pesquisada deverá estar envolvida por duas barras. Veja abaixo o resultado obtido com a execução do programa acima:

```
fooey         555-1234     2400/1200/300     B
foot          555-6699     1200/300          B
macfoo        555-6480     1200/300          A
sabafoo       555-2127     1200/300          C
```

Um exemplo com duas regras de pesquisa

O utilitário `awk` lê uma linha de cada vez do arquivo de entrada.

Para cada linha, ele testa os padrões de cada regra. Se vários padrões coincidem então várias ações são executadas na ordem em que elas aparecem no programa. Se nenhum padrão coincide, nenhuma ação é executada.

Após processar todas as regras coincidentes na linha, o `awk` lê a próxima linha. Isso continuará até que o final do arquivo seja alcançado. Veja o exemplo:

```
/12/  { print $0 }
/21/  { print $0 }
```

O exemplo acima contém duas regras: uma procura pela string `12` e a outra pela string `21`. Se uma linha contém ambas as strings elas serão impressas duas vezes, uma para cada regra. Se você executar o exemplo acima atuando nos arquivos de dados de exemplos `BBS-list` e `inventory-shipped` - como mostrado a seguir:

```
$ awk '/12/ { print $0 }
     /21/ { print $0 }' BBS-list inventory-shipped
```

obteríamos a seguinte saída:

```
aardvark      555-5553     1200/300          B
alpo-net      555-3412     2400/1200/300     A
barfly        555-7685     1200/300          A
bites         555-1675     2400/1200/300     A
core          555-2912     1200/300          C
fooey         555-1234     2400/1200/300     B
foot          555-6699     1200/300          B
macfoo        555-6480     1200/300          A
sdace         555-3430     2400/1200/300     A
sabafoo       555-2127     1200/300          C
sabafoo       555-2127     1200/300          C
Jan  21  36  64 620
```

```
Apr  21  70  74 514
```

Note, no destaque acima, que a linha do arquivo `BBS-list` que contém a string **`sabafoo`** foi impressa duas vezes, uma para cada regra.

Programas `awk` executáveis

Uma vez que você aprendeu com usar o `awk`, pode ser que você queira escrever scripts que possam ser executados simplesmente digitando-se o nome do programa. Para que isso seja possível, você poderá usar a sintaxe do mecanismo de script, que consiste em colocar na primeira linha do programa os símbolos `#!`.

Por exemplo, você poderia criar um arquivo chamado `hello.awk`, contendo as seguintes linhas:

```
#!/usr/bin/awk -f
# Um programa AWK simples
BEGIN {
    print "Olá!!!, Usuario. Este é um programa em AWK."
}
```

Após tornar este programa executável (como comando `chmod`) você pode simplesmente digitar no prompt de comando do Unix:

```
$ ./hello.awk
Olá!!!, Usuario. Este é um programa em AWK.
$
```

Seria a mesma coisa que digitar **`awk -f hello.awk`**

Comentários também podem ser adicionados aos programas `awk`. Usa-se o símbolo numeral (**#**) para colocar comentários nas linhas desejadas, da mesma maneira como no Korn Shell. Isso facilitará a vida do programador, pois posteriormente quando for necessária uma manutenção, os comentários ajudarão a entender melhor a lógica que foi usada no desenvolvimento do programa, mesmo que não seja outro profissional fará a manutenção nele.

Caso deseje aprender mais sobre o `awk`, consulte o endereço citado no início do capítulo. Nele será encontrado o manual completo da linguagem, onde você verá que se trata de uma linguagem poderosa no auxílio ao tratamento de dados obtidos a partir de arquivos textos diversos.

Capítulo 9 – Comandos do Unix úteis para a programação shell-script

Serão apresentados neste capítulo alguns comandos do Unix úteis para a manipulação de dados, como por exemplo obter o tamanho de uma string, obter uma substring, posicionamento do cursor na tela, etc. São recursos úteis, presentes no Unix e que podem facilitar o desenvolvimento de Scripts-Shell, pois o Unix foi criado com a filosofia de que as ferramentas possam ser combinadas para a obtenção do resultado desejado.

Obtendo o comprimento de uma string - ${#VARIÁVEL}

Pode haver a necessidade de se precisar contar quantos caracteres existem em uma determinada string, principalmente quando trabalhamos com variáveis de ambiente. O comando **${#VARIÁVEL}** pode ajudá-lo nesta tarefa. Exemplo:

```
NOME="Rodivaldo Marcelo"
print - ${#NOME}
17
```

Um exemplo de uso em scripts é quando queremos checar se em uma determinada variável recebeu a quantidade de caracteres correta através do comando `read`. Veja o exemplo, em que é checado se a entrada do código de cliente é menor do que 1 ou maior do que 5:

```
while (( 1 == 1 )); do
    print -n "Entre com o código do cliente:"
    read CODICLI

    TAMCOD=${#CODICLI}
    if (( TAMCOD == 0 || TAMCOD > 5 ));then
       print - "Código ter entre 1 e 5 caracteres de tamanho...\a"
    else
       exit
    fi
done
```

Extraindo subcadeias de caracteres de uma string

Existem algumas maneiras de se extrair partes de uma string. Existem dois comandos no Unix que podem ser usados para isso. Um é o comando **expr substr** e o outro é o comando **cut**, sendo que este último apesar de ser usado para arquivos de dados, pode ser usado também para a extração de caracteres de uma string. Outra opção é pelo uso de substituição de variáveis.

O comando `expr substr` possui a seguinte sintaxe:

expr substr *"string" Início Fim*

onde:

string - Texto a ser extraído a string. Deve estar entre aspas duplas.

Início - Posição inicial onde será iniciada a extração.

Fim - Posição onde terminará a extração.

Também pode ser extraída parte de uma string usando substituição de variáveis (usando-se **{}**), porém atente-se para que **a posição inicial começa a contar em 0 (zero).**

Exemplo com **expr substr** e com substituição de variáveis:

```
$ CODCLI=12532256
$
$ expr substr "$CODCLI" 1 3
125
$
$ print - ${CODCLI:0:3}
125
$
$ expr substr "$CODCLI" 4 5
32256
$
$ print - ${CODCLI:3:5}
32256
$
```

Caso sua implementação de Unix não possua o comando `expr` com os argumentos mostrados acima, um pipe com o comando `cut` produz o resultado desejado.

A sintaxe do comando `cut`, usando-se em scripts para a extração de caracteres é:

```
print - string|cut –cInício-Fim
```

onde:

string - Texto a ser extraído a string. Pode ser uma variável ou um literal. Neste caso estar entre aspas duplas é opcional, pois a cadeia de caracteres será passada para o comando `cut` através de pipeline.

Início - Posição inicial onde será iniciada a extração. A opção `–c` é obrigatória para o comando `cut`.

Fim - Posição onde terminará a extração. Se for omitida, a extração será feita até o final da string.

Veja o exemplo anterior, agora usando o comando **cut**:

```
CODCLI=12532256
print - $CODCLI|cut -c1-3
125
print - $CODCLI|cut -c4-8
32256
```

Veja abaixo dois exemplos do mesmo programa, um com o comando `expr substr` e o outro com o comando `cut`. Os dois mostram o código do cliente em um banco, onde os 3 primeiros caracteres identificam a agência e os 5 últimos a conta corrente.

Exemplo 1 - usando o comando **expr**:

```
print -n "Digite o codigo do cliente: "
read CODCLI
```

```
CODAGEN=$(expr substr "$CODCLI" 1 3)
CODCONT=$(expr substr "$CODCLI" 4 5)

print -n "Codigo da agencia......: $CODAGEN"
print -n "Conta corrente.........: $CODCONT"
```

Exemplo 2 - usando o comando **cut**:

```
print -n "Digite o codigo do cliente: "
read CODCLI

CODAGEN=$(print - $CODCLI|cut -c1-3)
CODCONT=$(print - $CODCLI|cut -c4-)

print - "Codigo da agencia......: $CODAGEN"
print - "Conta corrente.........: $CODCONT"
```

Exemplo 3 - usando substituição de variáveis (primeiro caractere inicia-se em **0**):

```
print - "Digite o codigo do cliente: \c"
read CODCLI

CODAGEN=${CODCLI:0:3}
CODCONT=${CODCLI:3:5}

print - "Codigo da agencia......: $CODAGEN"
print - "Conta corrente.........: $CODCONT"
```

Como resultado, a execução do programa será exemplificado:

```
Digite o codigo do cliente: 12532256
Codigo da agencia......: 125
Conta corrente.........: 32256
```

Removendo espaços em branco - `tr -s " "`

Sempre nos deparamos com casos em que determinadas cadeias de caracteres possuem mais de um espaço em branco, como no caso de um arquivo de dados que podem conter vários campos em um registro, separados por espaços ou tabulações, porém há mais de um espaço entre os campos. Podemos resolver este problema com o comando `tr -s " "` - através de pipeline - que fará com que a string com excesso de espaços seja transformada em uma string com espaços simples. Veja:

```
REGISTRO="005-001     Jose Carlos         Sao Paulo       Brasil"
print - "$REGISTRO" # Com o uso de aspas mantem-se os espaços
005-001     Jose Carlos         Sao Paulo       Brasil
print - "$REGISTRO"|tr -s " "
```

```
005-001 Jose Carlos Sao Paulo Brasil
print - $REGISTRO # Sem o uso de aspas remove-se o excesso de espaços
005-001 Jose Carlos Sao Paulo Brasil
```

Nos exemplo de Shell-script abaixo, é feito um processamento de arquivo em que são removidos os espaços excessivos de todos os registros, mostrando no final, a quantidade de registros processados:

```
print - "Início do processamento do arquivo clientes.dat\a"
print - "Aguarde..."

touch dados.tmp

tr -s " " < dados.dat >>dados.tmp

CONTAREG=$(wc -l <dados.dat)

rm dados.dat
mv dados.tmp dados.dat

print - '-------------------------------------------------'
print - "Total de registros processados ==> " $CONTAREG
print - '-------------------------------------------------'
```

Trocando as letras - `tr 'a-z' 'A-Z'` e `tr 'A-Z' 'a-z'`

Outra utilidade do comando `tr` é a capacidade de fazer a transformação de letras maiúsculas para minúsculas e vice-versa. Poderá ser usado este recurso em caso seja necessário passar um arquivo por um tratamento de conversão de caracteres antes de ser submetido a um processamento.

Pode-se usá-lo tanto na leitura direta de um arquivo - através do redirecionamento de entrada/saída - como através de pipeline, seja através do comando `print -` ou do comando `cat` - como nos exemplos a seguir.

Exemplo 1 - criação do arquivo `dados.mai`, em maiúsculas, a partir do arquivo `dados.ori`:

```
tr 'a-z' 'A-Z' < dados.ori >dados.mai
```

Exemplo 2 - criação do arquivo `dados.min`, em minúsculas, a partir do arquivo `dados.ori`:

```
tr 'A-Z' 'a-z' < dados.ori >dados.min
```

Exemplo 3 - conversão de uma string contendo letras minúsculas em maiúsculas usando `echo`:

```
TEXTO='esta variavel possui uma String com letras minusculas'
TEXTO=$(print - $TEXTO|tr 'a-z' 'A-Z')
print - $TEXTO
ESTA VARIAVEL POSSUI UMA STRING COM LETRAS MINUSCULAS
```

Observe o programa abaixo, onde é feita a conversão de letras minúsculas para maiúsculas e, ao final, é mostrada a quantidade de registros processados:

```
print - "Convertendo todas as letras para maiusculas... Aguarde"

tr "[a-z]" "[A-Z]" < cobranca.dat >cobranca.tmp

rm cobranca.dat
mv cobranca.tmp cobranca.dat

print - "Total de registros: $(wc -l <cobranca.dat)"
```

Para mais detalhes sobre as opções de transformação de caracteres, recorra ao manual on-line do comando **tr**.

Colocando o cursor em uma determinada posição - `tput cup`

Quem pensa que não é possível, no Unix, fazer um Shell-script apresentável, onde se pode posicionar uma mensagem em qualquer parte da tela, pode até se impressionar, pois existe o comando **tput cup** que permite fazer isso, não ficando o programador restrito apenas às estruturas de controle e sequencias de comandos.

O comando **tput cup** possui a seguinte sintaxe:

tput cup *Lin Col*

Lin - linha da tela onde será posicionado o cursor. Vai de 0 a 79.

Col - coluna da tela onde será posicionado o cursor. Vai de 0 a 24.

O exemplo a seguir ilustra um menu de duas opções onde a mensagens e textos em tela estão posicionados de maneira a deixar a execução do script mais apresentável e profissional:

```
# Menu de opcoes do sistema
# -------------------------
while true;do
    clear

    tput cup 02 15; print - "MENU DE OPCOES"

    tput cup 06 15; print - "d - mostra data e hora"
    tput cup 08 15; print - "w - para mostrar usuários logados"
    tput cup 10 15; print - "l - para listar o conteúdo do diretório
corrente"

    tput cup 20 15; print - "Por favor entre com sua opcao - ENTER
para sair: \c"
    read OPCAO
```

```
    case $OPCAO in
         [dD]) clear
               date
         ;;
         [wW]) clear
               who
         ;;
         [lL]) clear
               ls -l
         ;;
            *) exit
         ;;
    esac

    sleep 3
done
clear
```

Definindo um formato de saída para os dados - comando `printf`

Algumas formatações de dados também podem ser efetuadas com comandos do Unix em Scripts-Shell. Pode-se fazer alinhamento à esquerda, direita e inclusive, em alguns tipos de arquivos de dados, preencher strings (códigos ou números) com zeros à esquerda por exemplo.

Para que possamos obter esses resultados na formatação de dados, usaremos o comando `printf`. Para os que estão acostumados com a linguagem C, não sentirão dificuldade no seu uso, pois são usados grande parte dos comandos de formatação da função `printf()` da linguagem C.

Cobriremos as formatações que permitem alterar o alinhamento do texto (esquerda ou direita) e o preenchimento de strings com zeros à esquerda. O comando `printf` não faz mudança automática de linha após a exibição do valor a ser formatado. Se isso for necessário, no final da formatação deve-se colocar o caractere de controle `\n`, que tem a função de mudança de linha (newline). Para mais detalhes sobre outros formatos possíveis, consulte o manual online do comando `printf`.

Alinhando strings à direita ou à esquerda

O comando `printf` pode ser usado para alterar o alinhamento na apresentação de uma string, seja ela literal ou variável de ambiente, numérica ou não. Com relação a valores numéricos, geralmente eles ficam melhores se alinhados à direita, enquanto outros tipos de strings ficam mais bem apresentados se alinhados à esquerda.

A sintaxe do comando `printf`, para tratamento de alinhamentos, é a seguinte:

`printf "%Ns" valor/string` - alinhamento à direita

`printf "%-Ns" valor/string` - alinhamento à esquerda

onde:

N - é o tamanho que deverá ter o elemento a ser alinhado. Se for menor do que a quantidade de caracteres definida em *N*, será preenchido por espaços em brancos. Exemplos:

```
NOMECLI='Coca-Cola Inc.'
FATUCLI='2556779,32'

printf "%-20s\n" "$NOMECLI" #20 caracteres (alinhado a esquerda)
Coca-Cola Inc.
printf "%20s" "$NOMECLI" # 20 caracteres (alinhado a direita)
      Coca-Cola Inc.

printf "%-15s\n" $FATUCLI #15 caracteres (alinhado a esquerda)
2556779,32
printf "%15s\n" $FATUCLI # 15 caracteres (alinhado a direita)
     2556779,32
```

Preenchendo strings com zeros à esquerda

Outra opção de formatação muito útil do comando **printf** é a capacidade de preencher uma string, independentemente de ser numérica ou não, com zeros à esquerda. Geralmente essa capacidade será útil com string composta de números, como em arquivos textos que deverão ser processados por outros programas ou serem enviados por teleprocessamento, sendo que os aplicativos que tratarão esses números podem exigir que no lugar dos espaços em branco à esquerda do número, sejam colocados zeros para preenchimento desse espaço.

A sintaxe usada para o comando printf para efetuar o preenchimento de zeros a esquerda é:

```
printf "%0Nd" valor/string
```

onde:

N - é o tamanho que deverá ter a string. Se a `valor/string` for menor do que o especificado em *N*, o espaço excedente será preenchido por zeros. Exemplo:

```
VALOR='25896544'
printf "%012d\n" $VALOR # 12 caracteres (completados com zeros)
000025896544

SALDO='2500'
printf "%015d\n" $SALDO #15 caracteres (completados com zeros)
000000000002500
```

Efetuando uma pausa no processamento do script - `sleep`

Nos capítulos anteriores há exemplos do comando **sleep**. Este comando tem a função de dar uma pausa no processamento durante um determinado números de segundos. Logo após o processamento volta a continuar.

Este comando é bastante útil no caso de se mostrar mensagens durante um determinado tempo para que o usuário possa ler, ou então quando queremos que um processamento fique em execução dentro de uma estrutura de loop, mas queremos controlar de quanto em quanto tempo o processamento do loop deve ser retomado. Sua sintaxe é simples:

sleep *segundos*

No exemplo abaixo, o administrador do sistema (root) envia uma mensagem a cada 1 minuto para todos os usuários, durante 10 minutos, avisando que o servidor será desligado. Após o final da contagem, é executado o comando `shutdown -h`:

```
CTMIN=0
while (( CTMIN <= 10 ));do
    print - "O servidor saira do ar para manutencao ! ! !"|wall
    print - "Por favor encerrem seus trabalhos e deem logout..."|wall

    (( CTMIN = CTMIN + 1))
    sleep 60
done
shutdown -h now #Argumentos do comando shutdown podem variar
```

Capítulo 10 - Exemplos de Scripts-Shell úteis

Terminando essa obra, quero deixar para você exemplos de alguns scripts que poderão facilitar a vida do usuário, principalmente do administrador de sistema que às vezes precisa de ferramentas para gerenciamento e não tempo de desenvolvê-las.

Espero que eles sirvam de base para o seu contínuo aprendizado, assim como para sua análise das técnicas e lógica neles utilizadas. Não se preocupe se alguns dos scripts apresentarem algum erro de sintaxe em comandos ou mesmo de lógica. O leitor pode até me falar mal um pouco por isso, mas a versão funcionando eu testei e sei que trabalha, mas na obra nem todos eles estarão operacionais em 100% justamente porque quero que o leitor aprenda com os erros encontrados. Minha recomendação, como a de vários profissionais experientes na área de tecnologia da informação é:

NUNCA EFETUE TESTES DE PROGRAMAS EM AMBIENTES DE PRODUÇÃO E NEM USE O AMBIENTE DO CLIENTE COMO LABORATÓRIO PARA SEU APRENDIZADO POIS, ISSO PODE COLOCAR EM RISCO O ANDAMENTO DE UM NEGÓCIO OU MESMO CUSTAR SEU EMPREGO. ATENTE-SE PARA AS NORMAS DA COMPANHIA EM QUE TRABALHA, PRINCIPALMENTE AS DE SEGURANÇA.

Serão encontrados comandos que não foram vistos nesta obra. Detalhes sobre eles poderão ser pesquisados no Manual de Referência do Unix (comando **`man`**).

No final do livro serão encontradas respostas sugeridas para os exercícios propostos. Estou colocando desta maneira, respostas sugeridas, porque, na programação Shell assim como em quase todas as linguagens de programação, não há somente uma maneira de realizar uma determinada tarefa. Isso dependerá bastante de sua criatividade e, principalmente, dedicação. Não espere dominar a programação Shell do dia para a noite. A simplicidade sempre torna mais fácil e produtiva a criação e manutenção tanto de programas Shell quanto de qualquer outro programa em qualquer outra linguagem. Execute tarefas complexas, mas com instruções simples. Como dito pelos americanos, use o método KISS:

Keep **I**t **S**imple **S**tupid

Brincadeiras à parte, espero estar contribuindo para que sua vida profissional com o Unix se torne mais fácil.

Abraços a todos.

Limitando a quantidade de sessões simultâneas dos usuários - `unico_unico.sh`

Este script visa limitar cada usuário a ter somente uma sessão de terminal ativa. Os usuários que precisarem ter mais de uma sessão de login deverão estar cadastrados no arquivo de dados (arquivo tipo texto) `usuarios.podem.logar`. Cada linha deverá ter somente um logname e, separado por dois-pontos (`:`) um número indicando a quantidade máxima de sessões simultâneas que um usuário poderá ter. Isso ajuda o administrador de sistema no controle de usuários que necessitam de mais de uma sessão de terminal para trabalhar, evitando o consumo excessivo de recursos do sistema operacional com usuários em estado ocioso (*idle status*).

Estrutura genérica do arquivo `/etc/usuarios.podem.logar`

Nome_do_Usuário:*Número máximo de sessões de terminal*

```
# ------------------------------------------------
# Programa unico_login.sh
# Procura a ocorrência de $LOGNAME no arquivo de dados
# ------------------------------------------------
# Script para controle de quantidades de usuarios
# que podem ter mais de 1 login no sistema operacional
#

USUARIO=$(grep ^$LOGNAME: /etc/usuarios.podem.logar|cut -d: -f1)
NMVEZES=$(grep ^$LOGNAME: /etc/usuarios.podem.logar|cut -d: -f2)
JALOGADO=$(who|tr -s " "|tr " " ":"|grep ^$LOGNAME:|wc -l)

if [[ "$USUARIO" != "$LOGNAME" && "$LOGNAME" != "root" ]]; then
   if (( JALOGADO > 1 )); then
      print - "*****************************************"
      print - "  Voce ja esta logado como $LOGNAME"
      print - "*****************************************"
      echo
      print - "Fechando conexao extra...."

      sleep 5
      exit
   fi
else
   if (( JALOGADO > NMVEZES )); then
      print - "*****************************************"
      print - " Voce ja esta logado como $LOGNAME"
      print - " Seu limite de conexoes e $NMVEZES"
      print - "*****************************************"
      print -
      print - "Fechando conexao extra...."

      sleep 5
      exit
   fi
fi
```

Acrescente ao arquivo `/etc/profile`, de preferência na primeira linha dele, a seguinte linha de comando:

```
. unico_login.sh
```

Calculando a diferença entre intervalos de tempo - `diftempo.sh`

Com este script, apresentado em forma de função a ser chamada por qualquer outro script, você poderá saber quanto tempo leva um determinado processamento. O resultado da função é mostrado no formato *HH:MM:SS*, podendo ser armazenado para uso pelo método de substituição de variável ou substituição de comando. Ela é composta das funções `segundos` e `diftempo`. A sintaxe desta função é:

`diftempo` *`Início Fim`*, onde:

`Início` - Variável onde estará armazenada a hora inicial do processo. Antes de começar o processo, armazene a hora inicial da seguinte maneira: `HORAINI=$(date +"%H:%M:%S)`.

`Fim` - Variável onde estará armazenada a hora final do processo. Após o término do processo, armazene a hora final da seguinte maneira: `HORAFIN=$(date +"%H:%M:%S)`.

```
# ------------------------------------------------
# Função segundos - transforma a hora em segundos
# ------------------------------------------------

segundos ()
{
   # ---------------------------------------------------
   # Se for passada hora em branco, considera meia-noite
   # ---------------------------------------------------
   TEMPO=$1
   if [[ "$TEMPO" == "" ]];then
      TEMPO="00:00:00"
   fi

   # -----------------------------------------
   # Separa hora, minutos e segundos em variáveis
   # -----------------------------------------
   VAR1=${TEMPO:0:2}
   VAR2=${TEMPO:3:2}
   VAR3=${TEMPO:6:2}

   # -----------------------------------------
   # Retorna a hora transformada em segundos para
   # a função diftempo convertendo de sexagesimal
   # para decimal
   # -----------------------------------------
   print "($VAR1*3600)+($VAR2*60)+($VAR3*1)"|bc
}


# ------------------------------------------------
# Função para calcular a diferença de tempo entre
# duas horas passadas como parâmetro
# ------------------------------------------------

diftempo ()
```

```
{
   # --------------------------
   # Total de segundos em um dia
   # --------------------------
   TOTSEGDIA=86400

   # ----------------------------------------
   # Transforma hora inicial e final em segundos
   # ----------------------------------------
   HORA1=$(segundos $1)
   HORA2=$(segundos $2)


   # ------------------------------------------------
   # Se passou da meia-noite na hora final, adiciona
   # $TOTSEGDIA à hora final
   # ------------------------------------------------
   if (( $HORA2 < $HORA1 ));then
      (( HORA2 =+ TOTSEGDIA ))
   fi

   # ------------------------------------------------
   # Calcula a diferença entre a hora inicial e final
   # ------------------------------------------------
   (( FINAL = HORA2 - HORA1 ))


   # ----------------------------------------
   # Converte de a diferença entre as horas para o
   # formato sexagesimal
   # ----------------------------------------
   (( VHORA = ( FINAL / 3600 ) % 24 ))
   VHORA=$(printf "%02s" $VHORA)

   (( VMINU = ( FINAL / 60 ) % 60 ))
   VMINU=$(printf "%02s" $VMINU)

   (( VSEGU = ( FINAL % 60 ))
   VSEGU=$(printf "%02s" $VSEGU)

   # ----------------------------------------
   # Exibe a diferença entre horas inicial e final
   # ----------------------------------------
   print "$VHORA":"$VMINU":"$VSEGU"
}
```

Abaixo mostro exemplo de como usar o script `diftempo.sh` para saber quanto tempo levará para serem removidos arquivos com o nome core de todos os diretórios onde forem encontrados, usando o comando `find`:

```
# -------------------
# programa delcore.sh
# -------------------
. diftempo.sh # Carrega o arquivo com as funções de cálculo
              # de diferença de horas

# Armazena hora inicial
HORAINI=$(date +"%H:%M:%S")

print "Removendo arquivos CORE do servidor... Aguarde\a"

# Procura e remove arquivos com o nome core
find / -name core -type f -exec rm -f {} ';' -exec print - {} removido
';'

# Armazena hora do término
HORAFIN=$(date +"%H:%M:%S")

print "---------------------------------"
print "Remocao de arquivos core terminada"
print "Tempo total do processo: $(diftempo $HORAINI $HORAFIN)
print "---------------------------------"
print
```

Criação de variáveis armazenando comandos de cores ANSI – `colors.sh`

Com este script, desde que seu terminal ou emulador de terminal tenha suporte para exibição de cores, é possível usar as variáveis por ele criada para que informações possam ser mostradas coloridas. As variáveis são usadas com o comando `echo`. O script deve ser executado para carregar as variáveis de cores como exemplificado abaixo:

```
. colors.sh
```

Exemplo de uso das variáveis de cores, exibindo um texto em fundo vermelho com o texto em cor de frente amarela:

```
print - "$BG_RED$FG_YELLOW"
print - "Teste de mensagem amarela com fundo vermelho"
print - "$NORMAL_FGBG"
```

Este é o código do programa a ser carregado antes do uso das variáveis:

```
#!/usr/bin/ksh)
#
#------------------
# Cores de frente
#------------------
FG_BLACK=$(print "\033[30m")
FG_RED=$(print "\033[31m")
FG_GREEN=$(print "\033[32m")
FG_YELLOW=$(print "\033[33m")
FG_BLUE=$(print "\033[34m")
FG_MAGENTA=$(print "\033[35m")
FG_CYAN=$(print "\033[36m")
FG_WHITE=$(print "\033[37m")

#------------------
# Cores de Fundo
#------------------
BG_BLACK=$(print "\033[40m")
BG_RED=$(print "\033[41m")
BG_GREEN=$(print "\033[42m")
BG_YELLOW=$(print "\033[43m")
BG_BLUE=$(print "\033[44m")
BG_MAGENTA=$(print "\033[45m")
BG_CYAN=$(print "\033[46m")
BG_WHITE=$(print "\033[47m")

#------------------------
# Reset para a cor padrao
#------------------------
NORMAL_FGBG=$(print "\033[0m")
```

Pontuando valores maiores que 999 - `pontuar.sh`

Leia este número rapidamente: 1225698836989.44.

É bem mais difícil do que ler o mesmo número com a pontuação: 1,225,698,836,899.44.

O script mostrado abaixo tem a função de pontuar números acima de 999, colocando vírgulas a cada grupo de três dígitos, mantendo o ponto decimal caso exista.

```
#!/usr/bin/ksh
# Pontua numeros maiores que 999 usando vírgula
# Mantém o ponto decimal caso exista
pontuar ()
{
   VALOR_INI=$1
   if [[ "$VALOR_INI" == "" ]];then
      VALOR_INI='999999999.99'
   fi

   HA_PONTO=$(print - $VALOR_INI|awk '/\./ {print $1}'|wc -l)
   if (( HA_PONTO == 0 )) ;then
      VALOR=$VALOR_INI
      P_DEC=""
   else
      VALOR=$(print $VALOR_INI|cut -d. -f1)
      P_DEC="."$(print $VALOR_INI|cut -d. -f2)
   fi

   TAMSTR=${#VALOR} # Tamanho da string numérica

   CTRCAR=$TAMSTR
   CTAPTO=0
   NUMPTO=""

   # Inverte a string colocando vírgula a cada 3 dígitos
   #
   while (( $CTRCAR >= 1 ));do
      NUMPTO=$NUMPTO$(print $(expr substr "$VALOR" $CTRCAR 1))

      (( CTRCAR =- 1 ))
      (( CTAPTO =+ 1 ))

      if (( CTAPTO == 3 && CTRCAR >= 1 ));then
         NUMPTO=$NUMPTO$(printf ",")
         CTAPTO=0
      fi
   done

   TAMSTR=${#NUMPTO}
   CTRCAR=$TAMSTR
   NUMERO=""

   # Inverte novamente a string, agora pontuada
```

```
   #
   while (( $CTRCAR >= 1 ));do
      NUMERO=$NUMERO$(print $(expr substr "$NUMPTO" $CTRCAR 1))

      (( CTRCAR =- 1 ))
   done
   NUMERO=${NUMERO}${P_DEC}

   print - $NUMERO # Mostra a string numérica pontuada
}
```

O script abaixo mostra, como exemplo, o comando `bdf` do HP-UX (`df -kP` no Linux, Solaris ou AIX – adapte o script de acordo com sua plataforma de sistema operacional) com os valores referentes aos file systems com pontuação, facilitando a leitura dos números.

```
# ------------------
# programa newbdf.sh
# ------------------

. pontuar.sh

# Cabecalho de colunas de tela
#
cabecalho ()
{
   clear

   tput cup $LIN $COL01; printf "Log.Volume"
   tput cup $LIN $COL02; printf "Tot. Size"
   tput cup $LIN $COL03; printf "Used Size"
   tput cup $LIN $COL04; printf "Free Size"
   tput cup $LIN $COL05; printf "P.Used"
   tput cup $LIN $COL06; printf "Filesystem"

   (( LIN = $LIN + 1 ))

   tput cup $LIN $COL01; printf "============"
   tput cup $LIN $COL02; printf "==========="
   tput cup $LIN $COL03; printf "==========="
   tput cup $LIN $COL04; printf "==========="
   tput cup $LIN $COL05; printf "===="
   tput cup $LIN $COL06; printf "================="

   LIN=02

}

COL01=00
COL02=13
COL03=25
```

```
COL04=37
COL05=50
COL06=57

LIN=00

cabecalho

# ---------------------------------------------
# Obtem dados do comando bdf (ou df -kP do Linux)
# ---------------------------------------------
for REGISTRO in $(bdf|grep -v Filesystem|tr -s " "|tr " " ":");do
    VOLLOG=$(print - $REGISTRO|cut -d: -f1)
    TAMTOT=$(pontuar $(print - $REGISTRO|cut -d: -f2))
    ESPUSO=$(pontuar $(print - $REGISTRO|cut -d: -f3))
    ESPLIV=$(pontuar $(print - $REGISTRO|cut -d: -f4))
    PERLIV=$(print - $REGISTRO|cut -d: -f5)
    PTMONT=$(print - $REGISTRO|cut -d: -f6)

    tput cup $LIN $COL01; printf $VOLLOG
    tput cup $LIN $COL02; printf "%11s" $TAMTOT
    tput cup $LIN $COL03; printf "%11s" $ESPUSO
    tput cup $LIN $COL04; printf "%11s" $ESPLIV
    tput cup $LIN $COL05; printf "%3s"  $PERLIV
    tput cup $LIN $COL06; printf $PTMONT

    (( LIN = LIN + 1 ))
    if [ $LIN -gt 20 ];then
       LIN=02
       clear
    fi
done

echo
```

Total de memória consumida por usuários – `chk_mem_by_user.sh`

Este utilitário, desenvolvido originalmente para AIX e podendo ser facilmente portado para qualquer outra plataforma Unix, mostra a quantidade de memória consumida pelos processos. Os valores são mostrados por usuários, totalizando o total de memória consumido:

```
#!/usr/bin/ksh
# -------------------------------------
# Calcula memoria em uso pelos processos
# correntes em Bytes por usuario em AIX
# Pode ser migrado para outros Unix com
# pequenas alterações no código para a
# plataforma específica desejada
# -------------------------------------

. pontuar.sh # Carrega o script de pontuação de número

check_end_user ()
{
   if [[ "$USER_NAME" != "$CURRENT_USER" ]] ;then
      print_user_bytes

      CURRENT_USER=$( print - $FILE_RECORD|cut -d: -f1)
      TOTBYTES=0
   fi
}

print_user_bytes ()
{
   print - "Username: $CURRENT_USER Bytes Used: $(popntuar $TOTBYTES)"
   print - '-----------------------------------------------'
}

TEMPFILE=/tmp/tempfile.$$

ps -ef -o user,vsz|grep -v USER|tr -s " "|tr " " ":"|\
cut -d: -f2,3|sort >$TEMPFILE

CURRENT_USER=$(line < $TEMPFILE|cut -d: -f1)
RECORDS_COUNT=$(cat $TEMPFILE|wc -l)
RECCOUNT=0
BYTESSUM=0

for FILE_RECORD in $(cat $TEMPFILE);do
    USER_NAME=$(print - $FILE_RECORD|cut -d: -f1)

    check_end_user

    USER_NAME=$(print - $FILE_RECORD|cut -d: -f1)
    MEMO_SIZE=$(print - $FILE_RECORD|cut -d: -f2)
```

```
    (( TOTBYTES =+ MEMO_SIZE ))
    (( RECCOUNT =+ 1 ))
    (( BYTESSUM =+ MEMO_SIZE ))

    if (( RECCOUNT == RECORDS_COUNT ));then
       print_user_bytes
    fi
done
print - "Total of used memory: $(punctuate $BYTESSUM)"
print - '-----------------------------------------------'

rm $TEMPFILE
```

Veja o exemplo da saída após a execução do script:

```
Amount of memory used by userid
-------------------------------
UserID   Used memory
-------- -----------
daemon       1,312 B
marcelo      4,404 B
root        22,644 B
user24       1,784 B
user8        1,712 B
-------- -----------
Total       31,856 B
```

Usuários em estado ocioso por mais de 5 minutos - `check_idle.sh`

O programa abaixo mostra quais usuários estão ociosos no sistema, ou seja, não foi pressionado nem mesmo a tecla ENTER por mais de 5 minutos. Caso seja maior ou igual a 10 minutos o tempo de ociosidade, a linha contendo o a informação do usuário é mostrada em vermelho.

É usado o script colors.sh para carregar as variáveis de cores (sequencias ANSI) usadas no script.

```
clear

. colors.sh # Generic script that
            # load color variables

WHO_ONLINE_FILE=who_online.dat

who -u|awk '{print $1","$6","$7","$4"/"$3","$5}' >$WHO_ONLINE_FILE

print - "------------ --------- --------- ---------------"
print - " Login Name  Idle time Shell PID Login Date/Time"
print - "------------ --------- --------- ---------------"

for REGISTRO in $(cat $WHO_ONLINE_FILE);do
    LOGIN_NAME=$(print - $REGISTRO|awk -F, '{print $1}')
    LOGIN_IDLE=$(print - $REGISTRO|awk -F, '{print $2}')
    if [[ "$LOGIN_IDLE" == "." ]];then
       LOGIN_IDLE='Active'
    fi

    LOGIN_SPID=$(print - $REGISTRO|awk -F, '{print $3}')
    LOGIN_DATE=$(print - $REGISTRO|awk -F, '{print $4}')
    LOGIN_TIME=$(print - $REGISTRO|awk -F, '{print $5}')

    MORE_THAN_10=$(print - $LOGIN_IDLE|awk -F: '{print $2}')
    if [ "$LOGIN_IDLE" = "old" -o "$MORE_THAN_10" -ge "10" ];then
       print -n "$FG_YELLOW$BG_RED" ; tput bold
       printf "%-12s %9s %9s %8s %6s\n" $LOGIN_NAME $LOGIN_IDLE
$LOGIN_SPID $LOGIN_DATE $LOGIN_TIME
    else
       print -n "$FG_YELLOW$BG_BLUE" ; tput bold
       printf "%-12s %9s %9s %8s %6s\n" $LOGIN_NAME $LOGIN_IDLE
$LOGIN_SPID $LOGIN_DATE $LOGIN_TIME
    fi

    tput rmso
done

print - "------------------------------------------------"

LOGGED_USERS=$(wc -l <who_online.dat)
N_IDLE_USERS=$(grep -v ",.," <who_online.dat|wc -l)

print -n "Total of users in idle status: " ; printf "%17s\n"
$N_IDLE_USERS
```

```
print - "Total of logged users at $(date +%T): \c" ; printf "%13s\n"
$LOGGED_USERS
print - "-----------------------------------------------"

print - ""

rm $WHO_ONLINE_FILE
```

Apêndice A - Endereços Internet referentes a Unix – Jan/2022

Korn shell – AIX 7.2

https://www.ibm.com/docs/en/aix/7.2?topic=shells-korn-shell

https://www.ibm.com/docs/en/aix/7.2?topic=shell-enhanced-korn-ksh93

Manual da linguagem AWK

http://www.gnu.org/software/gawk/manual/gawk.html

Shell-script – algumas dicas e truques

https://www.profissionaloracle.com.br/2012/08/08/10-dicas-para-programar-em-shell-script/

http://www.unixmantra.com/2013/09/tips-tricks-to-be-comfortable-on-ksh.html

http://www.shelldorado.com/shelltips/

http://www.kingcomputerservices.com/unix_101/tips_on_good_shell_programming_practices.htm

https://www.ece.ucdavis.edu/~vojin/CLASSES/EEC70/W2004/UNIX-basics.pdf

https://www.dartmouth.edu/~rc/classes/ksh/print_pages.shtml

Apêndice B - Comandos Unix úteis em scripts

`bc` - Calculadora binária

Pode ser usada em modo interativo, assim como aceita que sejam passadas expressões via pipeline. Muito útil para fazer cálculos quando para fazer conversões de bases.

No modo interativo, digite `bc` no prompt de comandos do Unix e efetue cálculos normalmente. Seus operadores básicos são:

+ - Soma

- - Subtração

***** - Multiplicação

/ - Divisão

^ - Potência

() - parênteses para agrupamento

% - Módulo (resto da divisão)

`scale=`*N* - quantidade de casas decimais no resultado, onde *N* é o número de casas decimais.

`ibase=`*N* - Base numérica de entrada. *N* é a base desejada. O default é base 10.

`obase=`*N* - base numérica de saída. *N* é a base desejada. O default é 10.

É possível escrever programas parecidos com a linguagem C, envolvendo cálculos, para serem executados diretamente na calculadora `bc`. Também estão disponíveis várias funções como raiz quadrada, seno, tangente, etc. Veja o manual on-line para mais detalhes. Exemplos de uso no modo interativo:

```
scale=5       - 5 casas decimais
125/33
3.78788


ibase=10      - Entrada em base 10
obase=16      - Saída em base 16
9
9
10
A
13
D
15
F
16
10
```

```
2+3*5
17              - Ordem de cálculo obedece à regra matemática
(2+3)*5         - Usa-se parênteses para alterar a ordem do cálculo
25


3%2             - Resto da divisão os números
1
```

Veja agora os mesmos exemplos, agora usando o método de pipeline:

```
print - 'scale=5;125/33'|bc
3.78787

print - 'ibase=10;obase=16;9;10;13;15;16'|bc
9
A
D
F
10

print - '2+3*5;(2+3)*5'|bc
17
25

print - '3%2'|bc
1
```

O ponto-e-vírgula (`;`), como verificado acima, serve como separador de linhas, permitindo colocar-se mais de uma expressão em uma mesma linha.

`cut` - Exibição de campos ou colunas

Este comando geralmente é usado para exibir campos ou colunas em registros de arquivos. Porém, usando o método pipeline, pode ser usado para exibir substrings de expressões caractere.
Esta é a sintaxe para a exibição de colunas de arquivos é:

```
cut -cColunas arquivo
```

onde:

Colunas - são as colunas que devem ser exibidas das linhas do arquivo, pode ser especificadas listas de colunas individuais (separadas por vírgulas) ou faixas de colunas (especificadas por hífen). Podem ser combinados tanto listas de colunas como faixas.

Para a exibição de campos, utiliza-se a seguinte sintaxe:

```
cut -dDelimitador -fCampos arquivo
```

Delimitador - é o tipo de delimitador usado para separar os campos dos registros. Se não for informado, por default é considerado como sendo espaço ou tabulação.
Campos - são os elementos separados pelo delimitador especificado. Podem ser especificadas listas de campos (separados por vírgulas) ou faixas de campos (especificadas por hífen). Podem ser combinados tanto listas de campos como faixas.

Vale esclarecer que independentemente do tipo de extração usada (campos ou colunas), os dados serão exibidos na ordem em que os dados se encontram no arquivo, independente da ordem em que a lista de campos ou colunas seja especificada para as opções `-c` (para colunas) ou `-f` (para campos).

Veja alguns exemplos do uso do comando `cut`, usando campos ou colunas, pelo método de leitura de arquivo ou de um pipeline:

```
cut -d: -f1,3,7-9,13-15,21,22 dados.dos.clientes
```

O exemplo acima exibe os campos de ordem 1,3, de 7 ao 9, de 13 a 15, 21 e 22 do arquivo `dados.dos.clientes`, sendo que os registros do arquivo são separados por dois-pontos (`:`).

```
cut -c4-7,9,2,32-55 remessa.1000
```

No exemplo anterior são mostradas as colunas 2, da 4 até a 7, a 9 e da 32 até a coluna 55 do `arquivo remessa.1000`. Como dito anteriormente, apesar das colunas estarem sendo passadas fora de ordem para a opção `-c` do comando `cut`, elas serão apresentadas na sequência em que aparecem no arquivo de dados.

No exemplo a seguir é demonstrado o uso do comando cut com o método de pipeline para criar variáveis com partes do conteúdo da variável `REG`, que contém o registro armazenado, assim como é demonstrado o seu uso com o método de substituição de comandos:

```
REG='Clinton#Presidente dos EUA#55 anos#Hillary Clinton#Al Gore'

NOME=$(print - $REG|cut -d# -f1)
CARGO=$(print - $REG|cut -d# -f2)
```

```
ESPOSA=$(print - $REG|cut -d# -f4)

print - "O $CARGO $NOME é casado com $ESPOSA, tendo como vice"
print - "o Sr. $(print - $REG|cut -d# -f5)."
print - "Sua idade é $(print - $REGISTRO|cut -d# -f3)."
```

`expr length` - Retorna o tamanho de uma string

Este comando permite que você saiba o comprimento de uma string (literal ou variável) passada como parâmetro, isto é, a quantidade de caracteres contida na string. Nem todas as implementações Unix possuem o argumento length para o comando expr. Neste caso, coloque o conteúdo a ter seu tamanho medido em uma variável e use o comando `print - ${#VARIÁVEL}`. Exemplos:

```
expr length 'Curso de Programação Shell'
26
```

ou

```
TEXTO='Curso de Programação Shell'
print - ${#TEXTO}
26
```

```
AUTOR='José de Alencar'
print - "Tamanho da variável AUTOR: $(expr length "$AUTOR")
caracteres."
Tamanho da variável AUTOR: 15 caracteres.
```

ou

```
AUTOR='José de Alencar'
print - "Tamanho da variável AUTOR: ${#AUTOR} caracteres."
Tamanho da variável AUTOR: 15 caracteres.
```

Veja este exemplo de script onde uma mensagem será centralizada na tela:

```
# ------------------------------------------------------------
# Programa center.sh
# Centraliza mensagens na tela à partir da função center ()
# ------------------------------------------------------------

# ========================
# Função de centralização
# ========================
center ()
{
   LINHA=$1     # Linha onde será exibida o texto
   TEXTO=$2     # Texto a ser centralizado

   TAMAN=${#TEXTO} # Armazena tamanho do texto

   (( COLUNA = ( 80 - $TAMAN) / 2 )) # Calcula com base em
                                     # uma tela de 80 colunas

   tput cup $LINHA $COLUNA; print - $TEXTO # Exibe texto
```

```
   return
}

clear

center 20 'Este texto aparecerá no centro da tela...'
center 22 "Meu login name é $LOGNAME"
```

`expr substr` - Exibe partes de uma string

Este comando permite que sejam extraídos certa quantidade de caracteres de uma string (variável ou literal) à partir de uma determinada posição. Nem todos os Unix possuem o argumento `substr` implementado ao comando `expr`.

Sintaxe:

```
expr substr Texto Início Qtde
```

onde:

`Texto` - literal ou variável de onde será extraída a substring.

`Início` - posição da string onde será iniciada a extração.

`Qtde` - quantidade de caracteres a ser extraída

Veja o exemplo abaixo, onde é lido o arquivo `clientes.dat` de dados de 60 posições, onde nas 10 primeiras posições está o código do cliente, as 35 subsequentes o nome do cliente e as 15 restantes possui seu saldo. Serão lidos somente o código do cliente e o saldo. Esses dados serão gravados no arquivo `codsaldo.dat`, com os campos separados pelo caractere dois-pontos. No final é exibido em tela a quantidade de registros processados:

```
# --------------------
# Programa versaldo.sh
# --------------------

print - "Criando arquivo codsaldo.dat..."

TOTREG=0
for REGISTRO in $(cat clientes.dat);do
    CODCLI=$(expr substr "$REGISTRO" 1 10)
    SALCLI=$(expr substr "$REGISTRO" 46 60)

    print - ${CODCLI}:${SALCLI} >>codsaldo.dat

    (( TOTREG = TOTREG + 1 ))
done
print - '-----------------------------------------'
print - "Total de registros processados: $TOTREG"
print - '-----------------------------------------'
```

`printf` - Exibe dados (strings ou numéricos) formatados

Este comando, similar ao `printf` da linguagem C, pode ser usado para vários tipos de formatação de saída de dados. Mais detalhes podem ser encontrados no manual on-line (digitando `man printf`). Sua sintaxe básica é:

```
printf "formato" string
```

Pode-se usar vários formatos dentro das aspas, desde que elas sejam respectivas às strings ou variáveis passadas para serem formatadas. O comando `printf` não efetua a mudança de linha automática, sendo necessário colocar-se o caractere de controle `\n` no final do formato.
Abaixo mostro as três saídas de formatação que mais comumente uso para tratamento de dados e na formatação de suas saídas:

Alinhamento à direita: `printf "%Ns" string`, onde *N* é o tamanho que a string terá depois de formatada. Se for menor que *N* será completada com brancos.
Exemplo:

```
NOME='Marcelo'
CIDADE='Sao Paulo'
SISTEMA='Unix'

printf "O Sr. %11s mora em %15s usa %07s\n" $NOME $CIDADE $SISTEMA
O Sr.     Marcelo mora em       Sao Paulo e usa    Unix
```

Alinhamento à esquerda: `printf "%-Ns" string`
Veja o exemplo anterior, agora com a formatação alinhando à esquerda as variáveis passadas como argumento:

```
printf "O Sr. %11s mora em %15s usa %-7s\n" $NOME $CIDADE $SISTEMA
O Sr. Marcelo     mora em Sao Paulo       e usa Unix
```

Zeros à esquerda: `printf "%0Ns" Número Inteiro`, onde *N* é o tamanho da string resultando de *Número Inteiro*. Se *Número Inteiro* for menor do que *N*, será completada com zeros à esquerda.
Exemplo:

```
SALDOANT=1200
DEBITOS=1000
CREDITOS=500
(( SALDO = SALDOANT - DEBITOS + CREDITOS ))
printf "Voce tinha %06s, gastou %07s, ganhou %s08s \
       e agora possui %010s"


Voce tinha 001200, gastou 0001000, ganhou 00000500 e agora possui
0000000700
```

`set -u/+u` - Ativa ou desativa o uso de variáveis não existentes

`set +u` - faz com que as variáveis não inicializadas assumam o conteúdo NULL quando estas forem referenciadas na substituição de variáveis.

`set -u` - Se for usada esta opção, qualquer variável que não tiver sido inicializada gerará mensagem de erro quando for feita a substituição de variáveis usando o nome de uma variável inexistente.

Estas duas opções permitem que você controle a existência ou não de variáveis em seus scripts, pois em alguns casos você pode ou não necessitar de um script que valide a existência de variáveis, ou seja, se as variáveis chamadas foram ou não declaradas em um determinado ponto de seu Shell-script.
Exemplos:

```
set +u
print - CODIGO

set -u
print - CODIGO
Parameter not set
```

`sleep` - Faz uma pausa em segundos

Este comando tem a função de efetuar uma pausa durante alguns segundo no processamento, continuando o processo após esse tempo. Sua sintaxe é simples:

`sleep` *`Segundos`*, onde *`Segundos`* é a quantidade de segundos em que o processamento deve ficar parado. Após este período o processamento continua normalmente.
Exemplo:

```
print - 'Aguardando 5 minutos antes de continuar...'
sleep 300
print - 'Já se passaram os 5 minutos... Terminando o processo...'
sleep 2
clear
```

`tput cup` - Posicionamento do cursor em tela

Este comando tem a função de posicionar o cursor em uma determinada coordenada da tela. É bastante útil quando queremos posicionas uma determinada mensagem na tela. Sintaxe:

`tput cup` *`Linha Coluna`*`, onde:`

`Linha` - posição na tela que vai de 0 a 24
`Coluna` - posição na tela que vai de 0 a 79

Neste exemplo, a mensagem `Processando Dados` é colocada na tela iniciando na linha 10 e coluna 13:

```
tput cup 10 13
print - 'Processando Dados'
```

`tput ed` - Limpa até o final da tela

Este comando fará com que a tela seja apagada da posição atual onde se encontra o cursor até o final da tela. Sintaxe:

```
tput ed
```

Supondo que o cursor foi posicionado na linha 5 e coluna 0, se quisermos limpar a tela deste ponto em diante, ou seja, até o final da tela, basta usar as seguintes instruções:

```
tput cup 05 00
tput ed
```

`tput el` - Limpa até o final da linha

Este comando fará um apagamento da posição atual onde se encontra o cursor até o final da linha onde ele se posiciona. Sintaxe:

```
tput el
```

Supondo que o cursor foi posicionado na linha 5 e coluna 7, se quisermos apagar deste ponto até o final da linha corrente, basta usar as seguintes instruções:

```
tput cup 05 07
tput el
```

`tput blink` - Coloca o cursor em modo piscante

Com este comando pode-se ativar o modo piscante do terminal para exibir mensagens piscando na tela. Por exemplo, para que a mensagem `Aguarde` fique piscando ao lado da mensagem `Processando Arquivo`, use a sequência de linha abaixo:

```
print - "Processando Arquivo...\c"
tput blink; print - "Aguarde"
```

Para voltar a tela ao modo normal, use o comando `tput reset`.

`tput bold` - Coloca o cursor em modo negrito

Este comando permitirá que os caracteres em tela sejam mostrados com maior destaque, isto é, acentuando o brilho dos caracteres exibidos. Por exemplo, para que a mensagem `Aguarde` fique com brilho mais intenso ao lado da mensagem `Processando Arquivo`, use a sequência de linha abaixo:

```
print - "Processando Arquivo...\c"
tput bold; print - "Aguarde"
```

Para voltar a tela ao modo normal, use o comando `tput rmso`.

`tput smso` - Ativa o modo vídeo reverso

Este comando permitirá que a tela seja colocada em vídeo reverso, ou seja, caracteres negros em fundo branco (fundo verde, dependendo da cor do terminal). Por exemplo, para que a mensagem `Aguarde` fique em vídeo reverso ao lado da mensagem `Processando Arquivo`, use a sequência de linha abaixo:

```
print - "Processando Arquivo...\c"
tput smso; print - "Aguarde"
```

Para voltar a tela ao modo normal, use o comando `tput rmso`.

`tput rmso` - Desativa o modo vídeo reverso

Este comando permite que a tela seja restaurada ao modo de caracteres normal. É usado para desativar os comandos `tput bold` e `tput smso`. Sua sintaxe resume-se a:

```
tput rmso
```

Para a desativação do comando `tput blink` use o comando `tput reset`.

`tr -s "  "` - Remove o excesso de espaços

Este comando servirá para que sejam removidos excesso de espaços, seja de uma string passada por pipeline ou de um arquivo a ser tratado pelo comando. Suas sintaxes são:

`print - String | tr -s "  "` - modo pipeline

`tr -s "  " <Arquivo` - tratamento de arquivo

`tr "[a-z]" "[A-Z]"` - Converte caracteres em maiúsculas

Com este comando pode-se fazer conversão de caracteres alfabéticos de minúsculas para maiúsculas e vice-versa. Sintaxes:

`print - String | tr "[a-z]" "[A-Z]"` - modo pipeline convertendo para maiúsculas.

`print - String | tr "[A-Z]" "[a-z]"` - modo pipeline convertendo para minúsculas.

`tr "[a-z]" "[A-Z]" <Arquivo` - tratamento de arquivo convertendo para maiúsculas.

`tr "[A-Z]" "[a-z]" <Arquivo` - tratamento de arquivo convertendo para minúsculas.

Apêndice C - Respostas dos exercícios propostos

Como disse anteriormente, este capítulo destina-se a dar sugestões de respostas aos exercícios, pois na maioria deles, pode haver mais de uma opção para que ele possa ser resolvido e chegar ao mesmo resultado. Portanto, se ao fazer este exercício você notar que poderá usar outra técnica para resolvê-lo, esteja à vontade para usá-la, pois assim você desenvolverá suas próprias técnicas de programação Shell-script, podendo até mesmo melhorar os exemplos didáticos aqui apresentados.

Exercícios do Capítulo 1

Exercício 1

```
alias cls='clear'
```

Exercício 2

Abra o arquivo `$HOME/.kshrc` com o editor `vi`. Acrescente a ele a seguinte linha:

```
alias cls='clear'
```

Adicione a seguinte linha de comando ao seu arquivo `.profile`:

```
export ENV=$HOME/.kshrc
```

Encerre a sessão de terminal e inicie uma nova.

Teste se o alias definido anteriormente está ativo digitando:

```
cls
```

Exercício 3

Coloque as seguintes linhas de comando no seu arquivo `.kshrc` criado no exercício anterior:

```
alias rm='rm -i'
alias ls='ls -lFi'
alias clear='sleep 5; clear'
```

Teste se as alterações de comandos efetuadas estão operacionais.

Exercícios do Capítulo 2

Exercício 1

```
TODAY=$(date +"%d/%m/%y")
print - $TODAY
```

Exercício 2

Supondo que você possua o usuário `user10`, digite a seguinte linha de comando para listar o seu diretório `$HOME` (`/home/user10`):

```
ls -l ~user10
```

Se o diretório permitir o acesso, o comando `pwd` deverá retornar `/home/user10`

Exercício 3

```
MYNAME='Rodivaldo'
print - $MYNAME
```

Exercício 4

Torne global a variável `MYNAME` para que outros subprocessos chamados à partir do corrente possam acessar a variável:

```
export MYNAME
```

Exercício 5

```
readonly TODAY
TODAY="Novo Conteúdo"
/bin/ksh: TODAY: is read only - Retorna erro na tentativa de alteração da variável
unset TODAY
/bin/ksh: TODAY: is read only - Retorna erro na tentativa de apagar a variável
```

Não será possível efetuar a exclusão desta variável usando o comando `unset` ou mesmo alterar seu conteúdo. Note que os comandos acima, executados após o comando `readonly`, retornarão mensagem de erro.

A única opção é deslogra-se do Unix para que a variável seja eliminada.

Exercício 6

```
export PS1=$(whoami)@$(hostname):'$PWD=> '
```

Navegue por outros diretórios e note a mudança automática do diretório corrente.

Note que para o prompt informar o diretório corrente, dispensando o uso do comando `pwd`, a variável `PWD` precisa estar incluída entre aspas simples. Incluída entre aspas duplas o diretório do prompt ficará estático independente do diretório para onde o usuário navegar.

Exercícios do Capítulo 3

Exercício 1

```
print - \$1 million dollars... and that\'s a very excellent bargain !
```

Exercício 2

a) Exemplo usando atribuição com aspas duplas:

```
LONG_STRING="\$1 million dollars... and that\'s a very excellent
bargain !"
```

b) Exemplo usando substituição de comandos:

```
LONG_STRING=$(print - \$1 million dollars... and that\'s a very
excellent bargain !)
```

Exercício 3

No comando `banner good day` são passados dois argumentos para o comando, tendo como resultado uma palavra abaixo da outra.

No comando `banner "good day"`, apesar de existirem duas palavras, é passado somente um argumento para o comando. Como as palavras estão circundadas por aspas duplas, o significado de separador de argumentos do espaço em branco é ignorado.

Exercício 4

O comando `print - '$ABC'` retorna a string `$ABC`, pois ela está circundada por aspas simples.

O comando `print - "$ABC"` retornaria o conteúdo da variável `ABC`.

Se a variável não existir e **`set -u`** foi digitado para ativar a validação de variáveis não declaradas, será mostrada a mensagem **`Parameter not set`**, caso contrário será impressa uma linha vazia.

Exercício 5

```
ASPA_DUPLA=\"
```

ou

```
ASPA_DUPLA='"'
print - $ASPA_DUPLA
```

Exercício 6

```
print - Este anula o significado do caractere após ele: \\
Este anula o significado do caractere após ele: \
print - Estes anulam o de qualquer um por eles envolvidos: "'   '"
Estes anulam o de qualquer um por eles envolvidos: '   '
print - E estes anulam todos, exceto '\, $, $(), ${} e "': '"   "'
E estes anulam todos, exceto \, $, $(), ${} e ": "   "
```

Exercícios do Capítulo 4

Exercício 1

```
#!/usr/bin/ksh
# ------------------
# Programa prog01.sh
# ------------------

clear

# Exibe mensagem de boas vindas
# -----------------------------
print - "Olá....Este é  programa $0"
print - '-------------------------------------------'

# Criação da variável MYNAME
# --------------------------
export MYNAME='Rodivaldo'

# Exibe o conteúdo de MYNAME
# --------------------------
print - "Conteúdo da variável MYNAME: $MYNAME"
print - '-------------------------------------------'

# Mostra a data e hora através dos formatos
# do comando date
# ---------------------------------------
print - Hoje é $(date +"%d-%m-%Y") e são $(date +"%H:%M:%S")
print - '-------------------------------------------'

# Mostra a listagem de usuários logados no momento
# ------------------------------------------------
print - 'Usuarios atualmente logados no sistema:'
who -u|more -e
print - '-------------------------------------------'
print - " "
```

Exercício 2

`export DATE_VAR=$(date)` - Atribuição interativa

Crie e execute o programa abaixo:

```
#!/usr/bin/ksh
# --------------------
# Programa exibedata.sh
# --------------------

clear
print - "Abaixo o conteudo da variavel DATE_VAR:"
print - $DATE_VAR
```

Exercício 3

`$# - 6`

`$3 - -d`

`$7 - NULL` - (não existe o sétimo parâmetro posicional)

`$* - abc def -d -4 +900 xyz`

`$0 - meu_script.sh`

Exercício 4

```
#!/usr/bin/ksh
# ---------------------
# Programa pedenome.sh
# ---------------------
clear

print - "Digite o seu nome e sobrenome: \c"
read NOME SOBRENOME
print – ""
print - "Exibindo Sobrenome e Nome"
print - "$SOBRENOME , $NOME" #Troca de ordens das variáveis
```

Exercícios do Capítulo 5

Exercício 1

```
X=ABC
[[ "$X" == "XYZ" ]]
print - $?
1
X=xyz
[[ "$X" == "XYZ" ]]
print - $?
1
X=XYZ
[[ "$X" == "XYZ" ]]
print - $?
0
```

Exercício 2

```
Y=56
(( Y <= 0 ))
print - $?
1
Y=-5
(( Y <= 0  && Y >= -2 ))
print - $?
0
Y=0
(( $Y <= 0 ))
print - $?
0
```

Exercício 3

```
#!/usr/bin/ksh
# ------------------
# programa negpos.sh
# ------------------
clear

print - "Digite um valor: \c"
read Y

if (( Y >= 0 ));then
   print - "sim - Positivo"
else
   print - "nao - Negativo"
fi
```

Exercício 4

```
#!/usr/bin/ksh
```

```
# -------------------
# Programa lincom.sh
# -------------------
clear

if (( $# != 3 ));then
   print - "Quantidade de argumentos invalida...\a"
   print - "Devem ser passados 3 argumentos. Foram passados $#"
else
   print - "Foram passados os argumentos $* para"
   print - "o programa $0"
fi
```

Exercícios do Capítulo 6

Exercício 1

```
#!/usr/bin/ksh
# -------------------
# Programa dignome.sh
# -------------------

clear

NOME=""

while [[ "$NOME" != "sair" ]];do
    print - 'Digite seu nome (digite sair para encerrar):'
    read NOME

    # Transforma o conteudo de nome em minusculas
    NOME=$( print - $NOME|tr 'A-Z' 'a-z')
done

print - "Programa terminado...\a"
```

Exercício 2

```
#!/usr/bin/ksh
# -----------------
# Programa soma6.sh
# -----------------
clear

SOMA_TOT=0 # Somatorio dos numeros
CONTADOR=1 # Contador de parcelas
PARCELAS=""

while (( CONTADOR <= 6 )) ;do
    print - "Digite o ${CONTADOR}o. valor: \c"
    read VALOR

    if (( CONTADOR == 1 )) ;then
       (( PARCELAS = VALOR ))
    else
       PARCELAS=$PARCELAS'+'$VALOR
    fi

    (( SOMA_TOT = SOMA_TOT + VALOR ))
    (( CONTADOR = CONTADOR + 1 ))
done

print - '-------------------------------------------------'
print - "$PARCELAS = $SOMA_TOT"
print - '-------------------------------------------------'
print - ""
```

Exercício 3

a) Usando o loop `while` para controlar a execução do menu:

```
#!/usr/bin/ksh
# ----------------------
# Programa menu_while.sh
# ----------------------

while true;do
    clear

    print - 'Menu de opcoes do sistema'
    print - '========================='

    print - 'd - Data do sistema operacional'
    print - 'h - Hora do sistema operacional'
    print - 'l - Usuarios logados no momento'

    print - " "

    print - "Digite a opcao desejada ou ENTER para sair => \c"
    read OPCAO

    case $OPCAO in
         d|D) date +"%d/%m/%y"
              sleep 5
              ;;
         h|H) date +"%H:%M:%S"
              sleep 5
              ;;
         l|L) who -u
              sleep 5
              ;;
           *) print - 'Encerrando programa... bye bye'
              sleep 5
              break
              ;;
    esac
done

clear
```

b) Usando o loop `select` para controlar a execução do menu:

```
#!/usr/bin/ksh
# ----------------
# Programa menu.sh
# ----------------

PS3='Digite a opcao desejada ou ENTER para sair: '
```

```
OP1="Data do sistema operacional"
OP2="Data do sistema operacional"
OP3="Usuarios logados no momento"
OPX="Encerrar o programa atual"

select OPCAO in "$OP1" "$OP2" "$OP3" "$OPX" ;do
    case $REPLY in
         1) date +"%d/%m/%y"
            sleep 5
            ;;
         2) date +"%H:%M:%S"
            sleep 5
                 ;;
         3) who -u
            sleep 5
                 ;;
         4) print - 'Encerrando programa... bye bye'
                 sleep 5
                 break
                 ;;
    esac
done

clear
```

Exercício 4

```
#!/usr/bin/ksh
# ---------------------
# Programa checaroot.sh
# ---------------------

while true;do
    ROOT=$(who |grep root|wc -l)

    if (( ROOT > 0 ));then
       print - "Usuario root encontra-se logado...\a"
       sleep 2
    fi
done
```

Para que o programa acima rode em retaguarda, digite seguinte linha de comando:

```
checaroot.sh &
```

Exercício do Capítulo 7

Exercício 1

```
#!/usr/bin/ksh
# --------------------
# Programa mensagem.sh
# --------------------

clear

trap '' 1 2 15

while (( 1 == 1 ));do
    print - "Enviando mensagem para a tela a cada 3 segundos...\a"
    sleep 3
done
```

Ao ser executado, o programa acima não responderá aos sinais `HUP`, `INT` e `TERM`, porém o sinal `QUIT` ainda está ativo. Para interromper o programa basta pressionar a combinação de teclas *CONTROL+*. Isso interromperá o script e poderá gerar um arquivo com o nome `core`.

Caso o programa tenha sido colocado em retaguarda, localize o programa na lista de processos do Unix:

```
ps -ef|grep mensagem.sh
```

Verifique o número do processo (segunda coluna da esquerda para a direita) e envie um sinal `QUIT` ou `KILL`:

```
kill -3 Número_do_Processo
```

ou

```
kill -9 Número_do_Processo
```

Exercício 2

Coloque as linhas de comando abaixo dentro de scripts chamados, por exemplo, `backup_tar.sh` e `restore_gzip.sh`. Dê permissão de execução aos scripts e os rode conforme mostrado abaixo:

```
nohup backup_tar.sh &
```

```
nohup restore_gzip.sh &
```